18.1　使用 Windows 事件日志

Windows 系统中包含多种事件日志，在 Windows XP 系统中包括 4 种主要的事件日志。即应用程序、系统、安全性及 Internet Explorer 日志，如图 18-1 所示。这样用户可以方便地从系统中提取和查看日志，从而了解系统的当前运行情况。

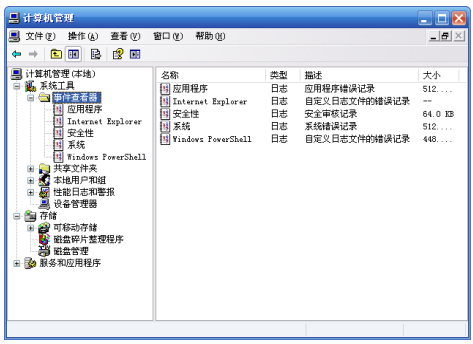


图 18-1　Windows XP 系统的事件日志

18.1.1　查看事件日志

在 Windows Vista 和 Windows Server 2008 版本中，事件日志的功能能够得到了大量改进和增强。通过使用 Get-EventLog cmdlet 可以查看事件日志内容，下面是脚本 GetEventLogs.ps1 的代码：

```
Get-EventLog –List
```

该脚本可获取本机中所有事件日志的清单，其中包括每个事件日志的大小、记录数，以及保存和覆盖策略等摘要信息，如图 18-2 所示。

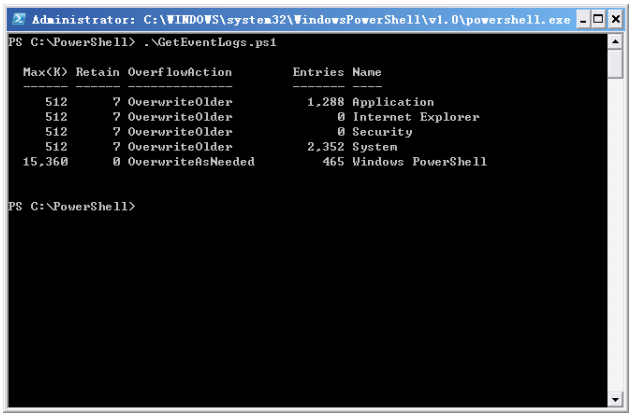


图 18-2　事件日志清单

18.1.2 读取事件日志

使用 Get-EventLog-list 查询当前计算机的事件日志后，可以使用 Get-EventLog 读取相应的日志，其基本形式是将事件日志的名称提供给 Get-EventLog cmdlet。GetApplicationEventLog.ps1 脚本的代码如下：

```
Get-EventLog application
```

运行该脚本显示所选日志的内容，如图 18-3 所示。

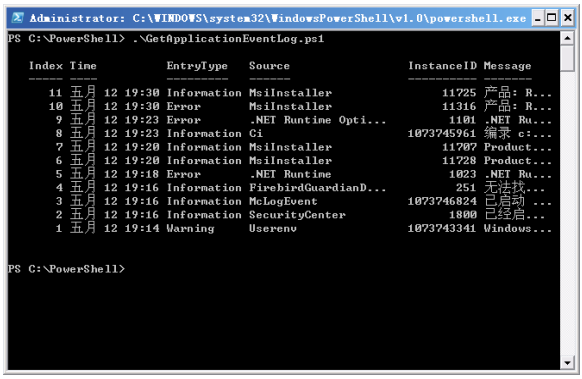


图 18-3 所选日志的内容

通常操作系统中有大量的系统日志，会显示为多屏。为了获取有用的信息，需要筛选输出内容。

（1）输出到文本文件

创建名为“WriteAppLogToText.ps1”的脚本输出查询结果到日志文本文件中，其代码如下：

```
Get-EventLog application >C:\PowerShell\Applog.txt
```

生成的文本文件内容如图 18-4 所示，其中保存完整的日志，随后即可查询其中关心的内容。

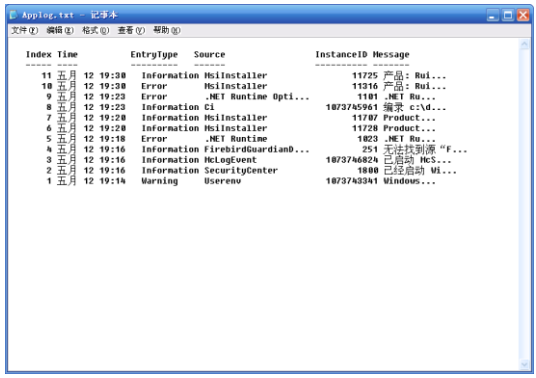


图 18-4 生成的文本文件内容

还可以通过 switch 语句和正则表达式搜索和提取所需的内容，脚本 switchAppTextLog.ps1 使用这种方法处理日志，其代码如下：

```
$AppLog = "C:\PowerShell\Applog.txt" $e=$i=$w=0
Switch –wildcard –file $AppLog{
"*error*" {$e++}
"*info*"  {$i++}
"*warn*" {$w++}
}
Write-Output "
$AppLog Contain following:
Errors $e
Warning $w
Information $i
"
```

执行结果如图 18-5 所示。

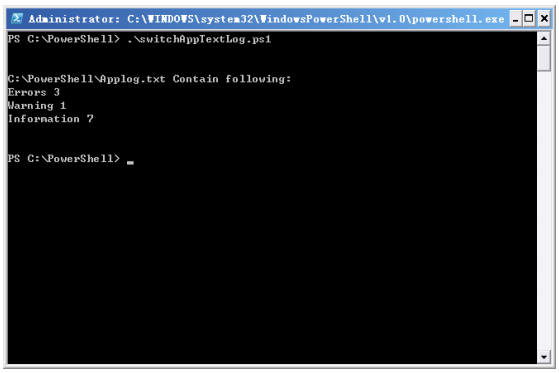


图 18-5　执行结果

（2）输出到 XML 文件

将日志导出为 XML 文件需要使用 Export-Clixml，这里使用 Get-EventLog cmdlet 指定要获取的事件日志的名称，用管道将日志结果传递给 Export-Clixml cmdlet。并使用 Export-Clixml cmdlet 的参数指定保存输出的 XML 文件路径，路径中包含的文件夹地址必须是现有存在的；否则将会发生错误，如图 18-6 所示。

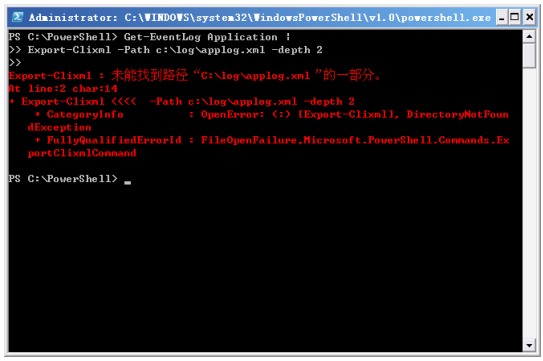


图 18-6　将系统日志输出到 XML 文件中出错

需要强调的是，在 Windows Vista 和 Windows Server 2008 中，没有提升权限的用户无法在系统分区根目录中写入文件。在这种情况下，需要具有操作对象的足够权限才能将日志文件保存在其中。如果要查看程序日志，则不必提升权限。在使用事件查看器（Eventvwr.exe）工具读取安全日志时，需要提升用户账号权限（UAC）。访问安全日志需要安全令牌中包括 seSecurityPrivilege 特权，默认情况下，只有管理员组的成员可以获得此权限，因此在需要将脚本提升权限后运行。最简单的方法是创建一个提升的 Windows PowerShell 的进程，为此单击鼠标右键后选择快捷菜单中的“运行方式”选项。在打开的对话框中输入管理员账号及密码，即可以管理员身份运行当前的 PowerShell 进程。

创建名为“WriteAppLogToXml.ps1”的脚本，将日志写入 XML 文件中，代码如下：

```
Get-EventLog application | Export-Clixml –Path C:\PowerShell\applog.xml –Depth 2
```

写入后可以使用 Excel 打开该文件。需要通过菜单中的“数据”|“XML”|“导入”选项选中其中要导入的文件，从弹出的“导入数据”对话框中选择“现有工作表中的 XML 列表”选项。随后 Excel 会用一段时间转换，表格中显示其中的内容。但是表中的列名称并不是事件日志中的字段名，这里的显示名称会类似“n”或“ns:1”，或其他名称。如果查看每一列数据的详细信息，则与日志文件中的数据相吻合，如图 18-7 所示。

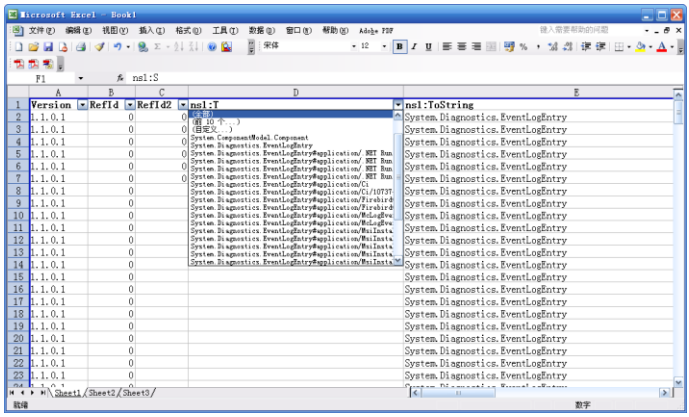


图 18-7　将 xml 格式的系统日志导入 Excel 中显示

18.1.3　写入事件日志

PowerShell 写入事件日志的功能很有用，Windows Vista 和 Windows Server 2008 的事件日志均提供了多种管理系统中发生事件的功能。通过使用.NET Framework 的 System.Diagnostics.EventLog 类，即可将信息写入到传统事件日志（即系统、应用程序，以及安全）中；另外还可以创建自己的事件日志，并将日志写入其中。

（1）创建来源

在写入事件日志之前，必须创建来源，来源可以被事件日志用于区别发生该事件的源头。这样如果需要写入共享来源的事件日志中，即可通过属性查询。在创建来源后，还需要为日志对象创建新的实例。将来源与事件日志关联，然后指定要写入的信息。

创建一个名为“WriteToAppLog.ps1”的脚本来实现这个功能，代码如下，后面将根据代码给出相关的说明：

```
if(![system.diagnostics.eventlog]::sourceExists("ps_script"))
{
$applog = [system.diagnostics.eventlog]::CreateEventSource("ps_script","Application")
}
$applog = New-Object system.diagnostics.eventlog("application",".")
$applog.source = "ps_script"
$applog.writeEntry("test for script write applog")
```

在该脚本中调用来源需要首先确定是否已经定义此来源并关联了事件日志，为此需要执行非运算。在来源没有定义的情况下进行容错处理，即在来源不存在的情况下创建一个指定来源名的来源。由于在脚本中用 system.diagnostics.eventlog 类的 SourceExist 静态方法判断来源是否存在，所以静态方法名前需要加两个冒号。如果来源不存在，则需要创建。在使用 CreateEventSource 方法时，必须指定来源及要关联到该来源的事件日志的名称。

在定义事件来源后，可以使用 New-Object cmdlet 创建事件日志的实例。在上面的脚本中，使用变量$applog 保存由 New-Object cmdlet 返回的时间对象名称。在调用该 cmdlet 时需要指定日志及其所在计算机的名称。在上面的脚本中，使用的是本计算机的日志。句点（.）表示当前计算机的快捷方式。

一旦在$Applog 日志中创建了到应用程序日志的引用，还需要使用 Source 属性将 ps_script 来源指定给该对象，然后使用 WriteEntry 方法将内容写入到应用程序日志。上面的代码执行后写入系统日志中的新记录如图 18-8 所示，在事件查看器中显示的内容图 18-9 所示。

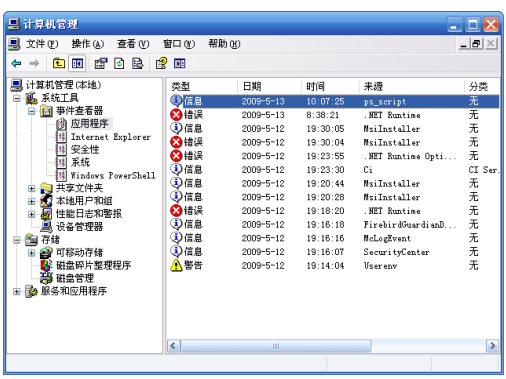


图 18-8　写入系统日志中的新记录

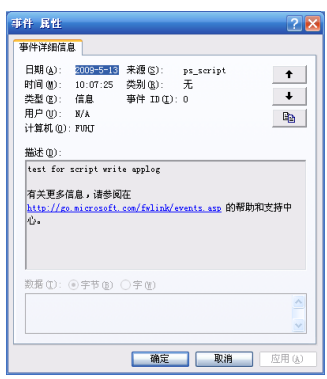


图 18-9　在事件查看器中显示的内容

（2）将 cmdlet 的输出内容保存到日志中

前面在日志中写入字符串也许看起来比较简单，但却有很多深层次的用途。如可以使用 WriteToAppLog.ps1 将脚本的运行情况及运行是否成功的结果写入事件日志，这对后面的程序排错和调试很有用。

另外还可以使用这种将信息写入到事件日志中的方法来保存 PowerShell 代码的运行结果。为了能够将 WMI 的查询结果写入到应用程序日志中，通过记录特定时间在特定计算机中运行的进程，则得到调整性能和增强安全性的第一手资料。创建名为“WriteProcessToAppLog.ps1”的脚本，其代码如下：

```
$strProcess = Get-WmiObject win32_process |
select-object name | Out-String
if(![system.diagnostics.eventlog]::sourceExists("ps_script"))
{
$applog = [system.diagnostics.eventlog]::CreateEventSource("ps_script","Application")
}
$applog = New-Object system.diagnostics.eventlog("application",".")
$applog.source = "ps_script"
$applog.writeEntry($strProcess)
```

该脚本使用 Get-WmiObject cmdlet 对 Win32_process 类执行基本的 WMI 查询，这样将会生成一个对象集合，其中包括本机中所有进程的属性和方法信息。为了减少返回的信息数量，可以使用 Select-Object 并选择需要的名称，这样获得的结果将是自定义的 Windows PowerShell 对象。为了将信息转换为字符串，需要使用 Out-String cmdlet。也可以使用名为“System.Diagnostics.EventLog”的.NET Framework 类的 WriteEntry 方法将包括在$strProcess 变量中的类表中的进程名称转换为字符串。

执行后写入的日志如图 18-10 所示。

图 18-10　执行后写入的日志

18.1.4　搜索事件日志

将事件日志导出为文本、XML 或其他格式后，搜索日志的最简单方法是使用 Get-EventLog cmdlet。不需要将日志保存为其他格式，而只需要用管道命令将结果传递给其他搜索的 cmdlet 即可，本节介绍一些技巧，首先创建一个名为“SearchByEventID.ps1”的脚本，其代码如下：

```
Get-EventLog –LogName system | Where-Object {$_.eventID –eq 1023}
```

要搜索事件日志，还需要知道 EventLogEntry 对象中的成员。该对象是名为“System.Diagnostics.EventLogEntry”的对象，并且是标准的.NET Framework 类。可以使用 Get-Member cmdlet 检索表 18-1 所示的 System.Diagnostics.EventLogEntry 对象的属性。

表 8-1　System.Diagnostics.EventLogEntry 对象属性

名　称	描　述
Category	System.String Category {get;}
CategoryNumber	System.Int16 Category {get;}
Container	System.ConponementModel.ICotainer Container{get;}
Data	System.Byte[] Data {get;}
entryType	System.Diagnostics.EventLogEntry EntryType{get;}
Index	System.Int32 Index {get;}
IndexID	System.Int64 InstanceId {get;}
MachineName	System.String MachineName {get;}
Message	System.String Message {get;}
ReplacementStrings	System.String[] ReplacementStrings {get;}
Site	System.String Site {get;}
Source	System.String Source {get;}
TimeGenerated	System.DateTime TimeGenerated {get;}
TimeWritten	System.DateTime TimeWritten {get;}
UserName	System.String Source {get;}
EventID	System.Object EventID {get=$this.get_EventID() –band 0xFFFF;}

（1）筛选属性

如果要减少 Get-EventLog cmdlet 返回的信息数量，可以用 Where-Object 过滤不需要的对象，事件日志中的常用筛选属性是 EventID、Source、Security 及 Message Text。

（2）选择来源

如果 Windows Installer（Windows 安装软件，包括适用于 32 位 Windows 操作系统的 Windows Installer 服务器，以及一个用于存储有关配置和安装信息的新软件包文件格式）出现问题，则可以在事件日志中查找所有来源名为“MsiInstaller”的事件。为此创建一个名为“FindMsiEvent.ps1”脚本查找所有有关 Msi 安装信息的日志，以在安装错误时解决问题，其代码如下：

```
Get-EventLog  application | where-object {$_.source -like "*Msi*"}
```

执行结果如图 18-11 所示。

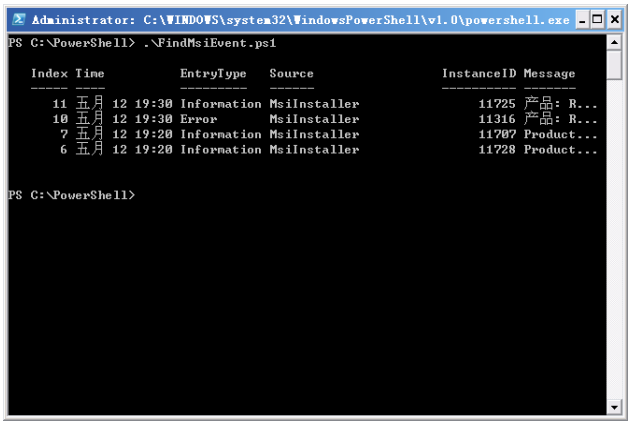


图 18-11　执行结果

（3）选择严重性

通常情况下用户可能只想看到“错误”级别的事件日志，为此创建一个 GetSystemLogError.ps1 脚本根据事件的严重性筛选系统日志。其中使用$StrType 变量保存要获取的事件日志项的类型名称，然后使用 Get-EventLog cmdlet 检索系统事件日志，并返回事件日志的对象集合。最后通过管道传递给 Where-Object cmdlet 过滤，过滤使用脚本中的$_变量值。使用默认视图输出该对象，代码如下：

```
$strLog = "system"
$strType = "error"
Get-EventLog $strLog | Where-Object {$_.entryType -eq $strType}
```

执行结果如图 18-12 所示。

（4）选择消息

搜索日志的另外一种方法是使用正则表达式匹配日志中的消息内容，可以通过 -match 参数使用 Where-Object cmdlet 调用正则表达式。

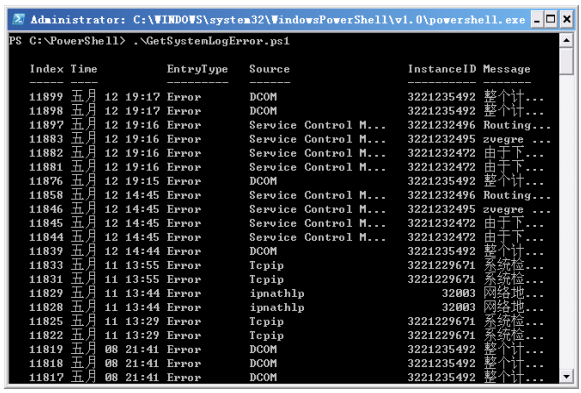


图 18-12　执行结果

用管道将 Get-EventLog cmdlet 生成的结果对象传递给 Where-Object cmdlet，为此使用自动变量$_，其中保存需要检索 Message 属性的事件日志项对象。随后使用 -match 参数配合 Where-Object cmdlet 查找包括在$strText 变量中的字符串，这里搜索的是有关 Firebird 数据库的所有信息。以下是 GetMessage.ps1 脚本的代码：

```
$strLog = "System"
$strText = "Firebird"
Get-EventLog -LogName $strLog | Where-Object {$_.message -match $strText}
```

执行结果如图 18-13 所示。

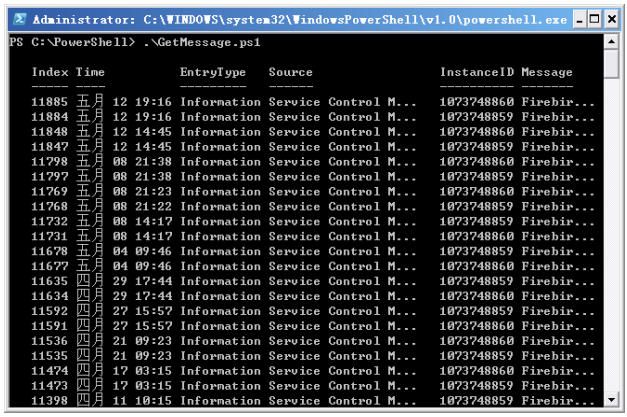


图 18-13　执行结果

18.1.5　管理事件日志

在使用事件日志时需要管理多个组件，其中最重要的是事件日志文件的大小。为了能够包括所需时间内的所有特定系统事件，通常情况下日志文件需要足够大，但是读取过大的文件将会耗费大量的时间。

（1）选择来源

使用事件日志时必须能够知道作为记录使用的日志，为此需要查看事件日志的源是否已经注册，实现这个目的的简单方法是使用 WMI 类 Win32_NtEventLogFile。创建名为“GetLogSources.ps1”的脚本，其代码如下：

```
$strLog = "Application"
Write-Host "The following sources are registered for the $strLog log: `n"
Get-WmiObject Win32_nteventlogfile -Filter "logfilename like '%$strLog%'" |
foreach { $_.sources }
```

上述代码针对应用程序日志，也可以用于其他日志。首先使用 Write-Host cmdlet 输出将要搜索的标题字符串，通过 Get-WmiObject cmdlet 查询 Win32_nteventlogfile WMI 类并筛选来源信息中包括事件日志名称的事件。最后使用管道传递给 ForEach-Object cmdlet 以逐个输出来源名称，执行结果如图 18-14 所示。

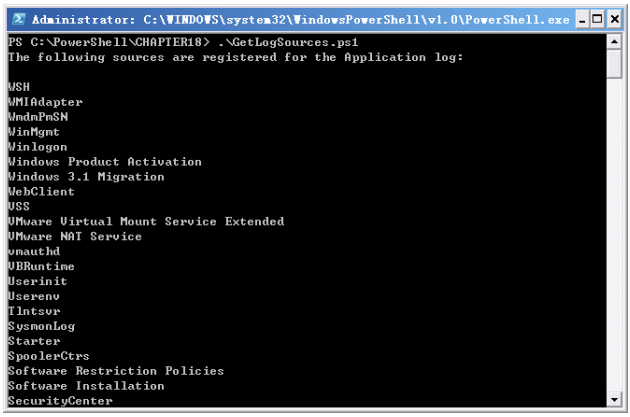


图 18-14　执行结果

（2）修改事件日志设置

存档策略对于系统日志很重要，不同策略决定留存的日志及其时间。根据系统的特点，不同日志分类的存档策略决定系统的稳定和安全。通常情况下，事件日志的最大值（以 KB 为单位）、最短保留期（以天为单位），以及其他各种策略都有相应的留存策略。在 Windows Vista 和 Windows 2008 中主要有如下 3 个可以设置的存档策略。

- DoNotOverwrite：当事件日志达到限制的最大值时保存现有日志项，此后的系统日志将不会再被写入而丢失。
- OverwriteNeeded：当事件日志满后添加项会依次替换最早的项目。
- OverwriteOlder：当事件日志满后新的项目将会覆盖比 MinimumRetentionDays 属性值定义的时间更早的项。如果事件日志已满，而且没有其他项比该属性值定义的时间早，则丢弃新项。

创建用于查询系统日志存档策略的脚本文件 GetEventLogRetentionPolicy.ps1，其代码如下：

例 18-1　创建用于查询系统日志存档策略的脚本文件

```
$strLog = "Application"
$objLog = New-Object System.Diagnostics.Eventlog("$strLog")
Write-Host `
 "
 The current settings on the $($objlog.LogDisplayName) file are:
 max kilobytes: $($objLog.maximumKiloBYtes)
 min retention days: $($objLog.minimumRetentionDays)
 overflow policy: $($objLog.overFlowAction)
"
```

其中使用 New-Object cmdlet 创建 System.Diagnostics.Eventlog 类的实例。在创建时需要传递指定参数用于确定需要处理的事件日志名，创建应用程序日志的对象后使用 Write-Host cmdlet 输出 LogDisplayName、maximumKiloBYtes、minimumRetentionDays 及 overFlowAction 属性。为了避免后 3 项只是简单地输出相关属性及其对象名称，为每个变量添加美元符（$）前缀并用括号分隔，如 $($objLog.maximumKiloBYtes)。

该脚本的执行结果如图 18-15 所示。

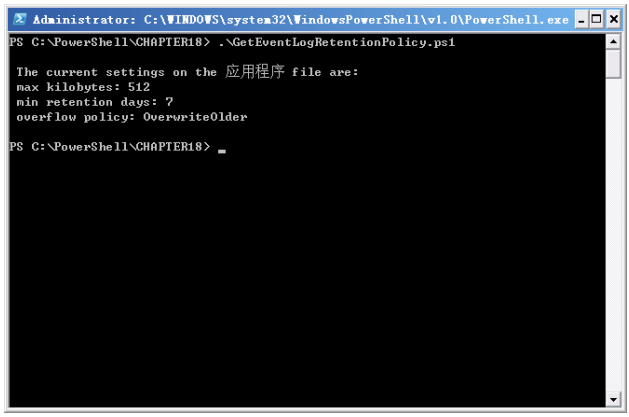


图 18-15　执行结果

更改事件日志存档策略时也需要使用 System.Diagnostics.Eventlog 类来实现，还需要指定将要配置的是存档策略。ModifyOverFlowPolicy 方法需要两个参数，即策略名和要使用的日期天数。在设置 DoNotOverwrite 或 OverwriteAsNeeded 时，忽略该方法调用的第 2 个参数。

创建脚本 SetEventLogRetentionPolicy.ps1 修改日志的存档策略，其代码如例 18-2 所示。

例 18-2　修改日志的存档策略

```
function DisplayLogSettings()
{
 Write-Host `
 "
 The current settings on the $($objlog.LogDisplayName) file are:
```

```
 max kilobytes: $($objLog.maximumKiloBYtes)
 min retention days: $($objLog.minimumRetentionDays)
 overflow policy: $($objLog.overFlowAction)
"
 if (!$args) { ChangeLogSettings("help") }
}
function ChangeLogSettings($policy)
{ if($policy -ne "help")
    {
     Write-Host -ForegroundColor green "changing log policy ..."
    }
 switch($policy)
  {
   "doNotOW"     { $objlog.modifyoverflowpolicy("DoNotOverwrite",-1) }
   "owAsNeeded" { $objlog.modifyoverflowpolicy("OverwriteAsNeeded",-1) }
   "owOlder"     { $objlog.modifyoverflowpolicy("Overwriteolder",$intRetention) }
   DEFAULT       {
                 Write-Host -ForegroundColor red `
                  "
                 You need to specify either of the following: `n
                 doNotOW - do not overwrite logs
                 owAsNeeded - overwrite as needed
                 owOlder - overwrite events older than $intRetention days `n
                 Example: > SetEventLogRetentionPolicy.ps1 doNotOW
                           Sets retention policy to Do not Overwrite
                 Example: > SetEventLogRetentionPolicy.ps1 owAsNeeded
                           Sets retention policy to Overwrite as needed
                 Example: > SetEventLogRetentionPolicy.ps1 owOlder
                           Sets retention policy to Overwrite older than 30 days
                 Example: > SetEventLogRetentionPolicy.ps1 help
                           Displays this help message
                  "
    exit             }
 }
}
$strLog = "application" #modify for different log
$intRetention = 30       #modify for different number of retention days
$objLog = New-Object system.diagnostics.eventlog("$strLog")
DisplayLogSettings($args)
ChangeLogSettings($args)
DisplayLogSettings($args)
```

注意在设置日志的存档策略时需要管理员权限；否则会出现错误，这是为了保护日志的有效性。如果低权限的非法用户试图恶意修改日志存档设置来逃避对操作的追查，则不可能。在脚本中初始化的变量$initRetention，默认值为 30 意味着事件日志存档策略会被设置为 30 天，当然只有在设置 OverwriteOlder 策略后才会生效。在脚本中使用$objLog 变量保存 System.Diagnostics.Eventlog 类的实例，并定义要设置的事件日志名称。在创建并初始化这些对象后，需要调用 DisplayLogSettings 函数。该函数可以以 KB 为单位显示事件日志的大小上限，以天为单位显示最小留存时间及其超时后的操作。

在输出当前时间存档设置后退出函数，并继续处理脚本中剩余的代码。如果用户在运行脚本时未提供相关的参数，则使用 help 参数调用 ChangeLogSetting 函数，以输出详细的帮助信息并退出整个脚本的执行。

这个脚本的关键部分是 ChangeLogSetting 函数，它使用 switch 语句控制程序的逻辑。允许用户设置事件日志存档策略的参数，而不需要记忆复杂的 System.Diagnostics.Eventlog 类的语法。该 switch 语句的输入参数保存在配置文件 $Profile 中，用户在输入时分别匹配参数-donotw、-owasneeded 及-owolder，分别对应于 DoNotOverwrite、OverwriteNeeded 和 OverwriteOlder 这 3 种存档策略。如果用户的输入未与其中的任何一个参数一致，则触发默认操作。这里的默认操作是用红色输出表现的帮助信息，这样可以帮助不熟悉此脚本的用户在执行出错后根据输出改进自己的输入。不带参数直接执行脚本的执行结果如图 18-16 所示，将存档策略由 OverwriteOlder 改变为 DonotOverwrite 后的执行结果如图 18-17 所示。

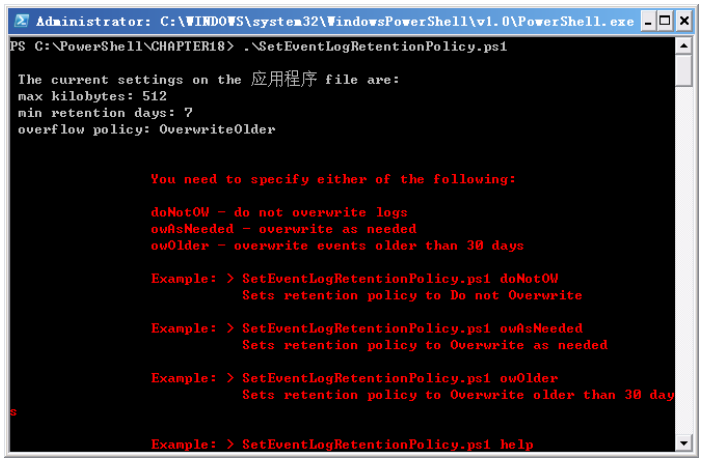


图 18-16　不带参数执行脚本的结果

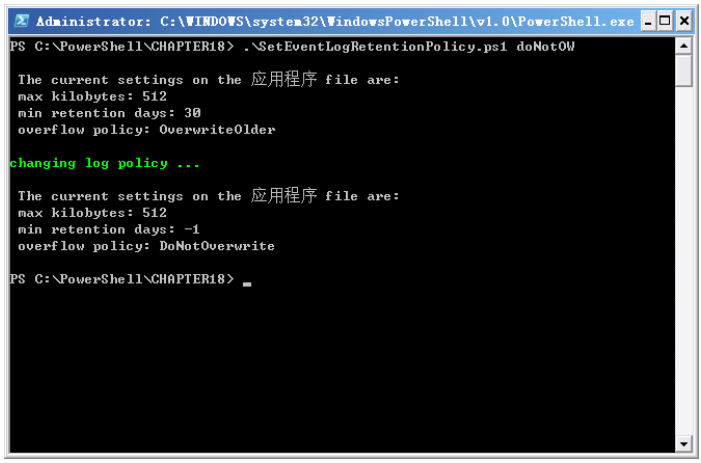


图 18-17　将日志存档策略改为 DonotOverwrite 后执行结果

18.1.6 创建事件日志

用户可以通过创建事件日志来跟踪常见的操作以维护系统，为此使用System.Diagnostics.EventLog 类的 CreateEventSource 方法，并提供事件日志的来源和名称。虽然一个事件日志可以保存多个来源的信息，但是一个事件来源只能关联一个事件日志。为了避免出错，需要使用 SourceExist 方法，并提供要查找的来源名称。如果来源不存在，则创建该来源和对应的事件日志；如果来源已经存在，则输出错误信息并退出脚本。CreateEventLog.ps1 脚本创建事件日志，如下例 18-3 所示。

例 18-3 创建事件日志

```
$strProcess = get-WmiObject win32_process |
  select-object name | out-string
$source = "ps_script"
$log = "PS_Script_Log"
if(![system.diagnostics.eventlog]::sourceExists($source,"."))
 {
  [system.diagnostics.eventlog]::CreateEventSource($source,$log)
 }
ELSE
 {
  write-host "$source is already registered with another event Log"
  EXIT
 }

$strLog = new-object system.diagnostics.eventlog($log,".")
$strLog.source = $source
$strLog.writeEntry($strProcess)
```

如果事件来源已经注册给不同的事件日志并需要该来源使用自定义的事件日志，则必须首先删除该事件来源，为此可以使用 system.diagnostics.eventlog 类的DeleteEventSource 方法。创建 DelectEventSource.ps1 脚本使用 SourceExist 方法判断事件来源是否已经被注册，如果已经注册，则使用 LogNameFromSourceName 方法输出该来源所指向的事件日志的名称。然后删除该事件来源，并传递已删除的消息；如果该事件来源尚未在系统注册，则会收到来源未注册的消息。该脚本的代码如下：

```
$source = "ps_script"
if([system.diagnostics.eventlog]::sourceExists($source,"."))
 {
  $log = [system.diagnostics.eventlog]::LogNameFromSourceName($source,".")
  Write-Host "$source is currently registered with $log log."
  Write-Host -ForegroundColor red "$source will be deleted"
 [system.diagnostics.eventlog]::DeleteEventSource($source)
 }
 ELSE
  { Write-Host -ForegroundColor green "$source is not regisered" }
```

执行以上两个脚本后添加的事件日志可以通过系统的事件查看器（运行eventvwr.exe）看到。

18.2　性能计数器

Windows 系统性能计数器是操作系统支持用于应用和组件发布性能数据的特殊对象，并且支持其他应用获取和分析这些已发布的数据。Windows 中的性能计数器很多，包括针对磁盘、网络和 TCP 等计数器，用户能够使用这些性能计数器提供的数据确定程序瓶颈和系统性能。通过运行 perfmon.msc 启动性能计数器数据，如图 18-18 所示。用户可以通过右击计数器清单添加其他计数器，在添加的过程中显示如图 18-19 所示的“添加计数器”对话框。

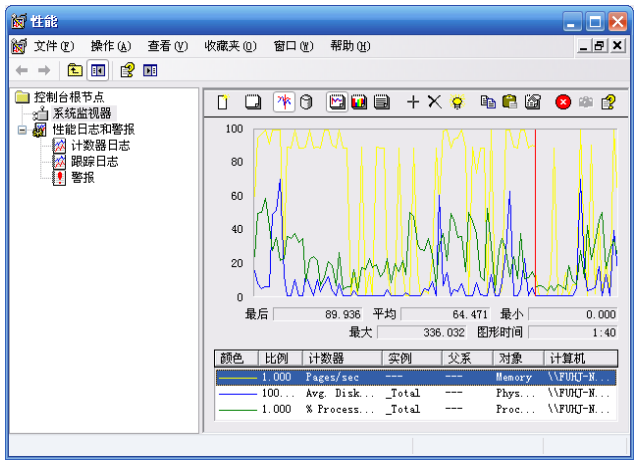


图 18-18　系统性能计数器

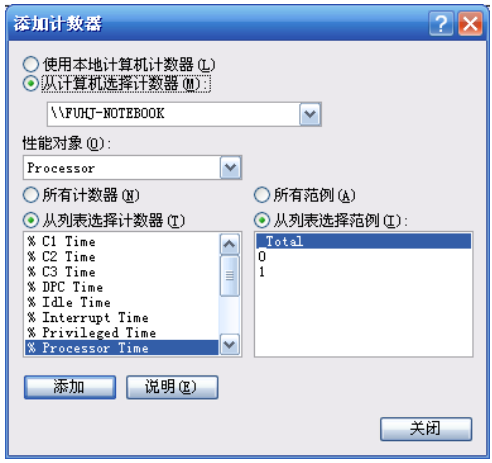


图 18-19　“添加计数器”对话框

其中包括性能计数器名，如%Processor Time 和 Handle Count 等，以及计数器对应的范例。当需要获取处理器相关的性能计数器数据时，默认以处理器命名。如果有多个处理器，则为计数器分别添加类似于#1、#2 和#3 这样的后缀。如果需要处理多个实例，则在添加计数器的过程中制定相应的实例。

18.2.1 Consuming Counter Data

PowerShell 没有内置的 cmdlet 用于获取性能计数器的值，所以必须使用.NET 的类。为此创建 System.Diagnostics.PerformanceCounter 的实例，配置其中的属性并不断调用 NextValue()方法来获取相应的值。在开始操作性能计数器之前，需要首先配置用户权限为管理员权限。然后创建脚本用于监视 CPU 的使用率，将脚本命名为“Monitor-CpuUsage.ps1”。将会用性能计数器获取并显示当前 CPU 每秒的占用率，这个脚本会一直循环执行下去；除非用户按 Ctrl-C 终止。其代码如下：

```
$counter = New-Object Diagnostics.PerformanceCounter
$counter.CategoryName = "Processor"
$counter.CounterName = "% Processor Time"
$counter.InstanceName = "_Total"
while ($true)
{
$value = $counter.NextValue()
Write-Host "CPU: $value"
sleep 1
}
```

其中涉及的计数器名和实例名都可以很容易地从“添加计数器”对话框中找到，Monitor-CpuUsage.ps1 脚本的执行结果如图 18-20 所示。

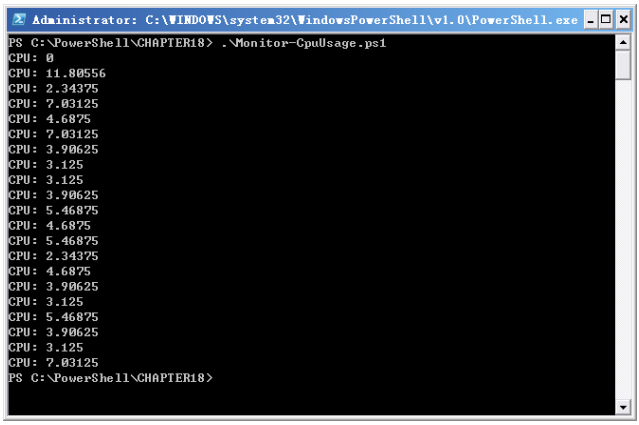


图 18-20 执行结果

从图中可以看到此时的 CPU 负载并不重，只达到 10%左右，用户可以用类似的方法获取其他性能计数器的数据。

18.2.2 监视程序

当使用程序时发生不正常的情况时，通常都会造成占用大量的系统资源而导致系统重启。这时使用性能计数器可以监控程序的运行情况，即在运行程序的同时启动监视脚本。该脚本会实时获取和分析性能数据，当发生异常时会向用户发出警告。

针对意外终止的应用程序创建一个监视脚本 Monitor-Crashes.ps1，该脚本将会启动外部进程。当程序意外退出时，脚本会将重启 5 次应用程序之后停止尝试，其代码如下：

```
function Start-Process
{
Write-Host "Starting process..."
.\UnpredictableCrash.exe
}
for ($i = 0; $i -lt 5; $i++)
{
Start-Process
Write-Host "Process exited $($i + 1) times."
}
Write-Host "Program restart limit exceeded."
```

该脚本的执行结果如图 18-21 所示。

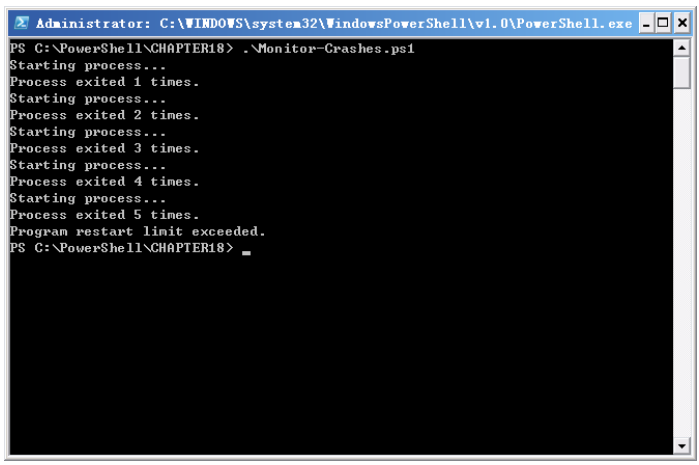


图 18-21　执行结果

在脚本执行过程中使用了一个名为“UnpredictableCrash.exe”的程序，这个程序执行一秒钟等待操作后并退出。通过这个方法来模拟执行一定的操作，但在特定条件下意外退出的程序。在脚本中只是重启程序 5 次。如果需要，则可以让程序一直执行，使其作为一个驻留程序来执行。

18.3　桌面计算机维护

从 Windows XP 开始，系统的默认设置对于普通用户已经足够，但是对于高性能计算或者高安全环境可能并不能满足用户的需求。为此需要通过监视系统中被修改的内容，并随着性能优化的需要调整。

18.3.1 维护计算机

在检查系统的驱动器配置时，首先应了解已安装的驱动器。因为 Windows 会将实际的驱动程序从物理驱动层抽象出来，因此用户可能不会意识到设备到底是使用了一个还是一组驱动。

（1）维护驱动设备清单

很多硬件供应商为了将操作系统的映像备份在硬盘中，会在硬盘中创建隐藏分区。通常厂商将系统的备份镜像放置其中，一般用户不会发现。因为没有为这个分区分配盘符，而且分区是隐藏的。创建名为“ReportDiskDriveConfiguration.ps1”的脚本，其中使用 Get-WmiObject cmdlet 和 Win32_DiskDrive WMI 两个类获取物理硬盘中的所有分区信息。该脚本可接收保存获取硬盘信息的远程计算机名的一个参数，或者问号（?）用于输出帮助信息。

例 18-4　获取分区信息

```
if(!$args)
 {
  Write-Host -foregroundcolor green `
  'Querying localhost ...'
  $args = 'localhost'
 }
if($args -eq "?")
 { "
    ReportDiskDriveConfiguration.ps1

DESCRIPTION:
    This script can take a single argument, computer name.
It will display drive configuration on either a local
or a remote computer. You can supply either a ? or a
name of a local machine.

EXAMPLE:
ReportDiskDriveConfiguration.ps1 remoteComputerName
reports on disk drive configuration on a computer named
remoteComputerName

The script will also display this help file. This is
done via the ? argument as seen here.
ReportDiskDriveConfiguration.ps1 ?
"
 }
Get-WmiObject -Class Win32_DiskDrive `
-computer $args
```

其中首先使用取反操作符(!)检查$args 变量是否存在，如果不存在，则执行默认操作，即搜索本地计算机的硬盘信息。并且将该变量值设置为 localhost，使用绿色信息输出结果，如图 18-22 所示。

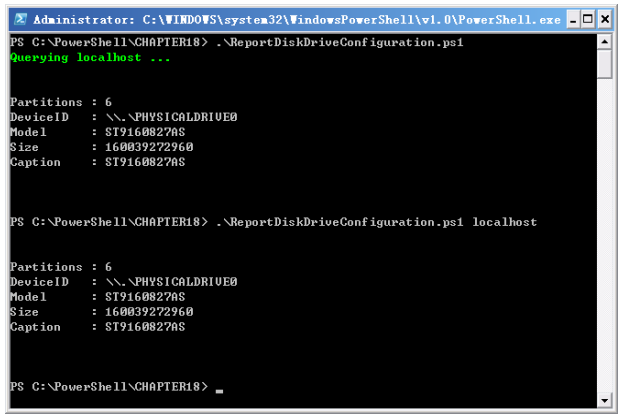


图 18-22　输出磁盘信息

（2）处理磁盘分区

PC 通常只会有一个磁盘，Windows 的管理策略会以分区形式管理磁盘，这样即可将物理硬件从操作系统中抽象出来。分区概念对于高效率地维护系统和文件很有好处，用户可以在“计算机管理”的“磁盘管理”工具中查看磁盘和磁盘分区之间的关系，如图 18-23 所示。

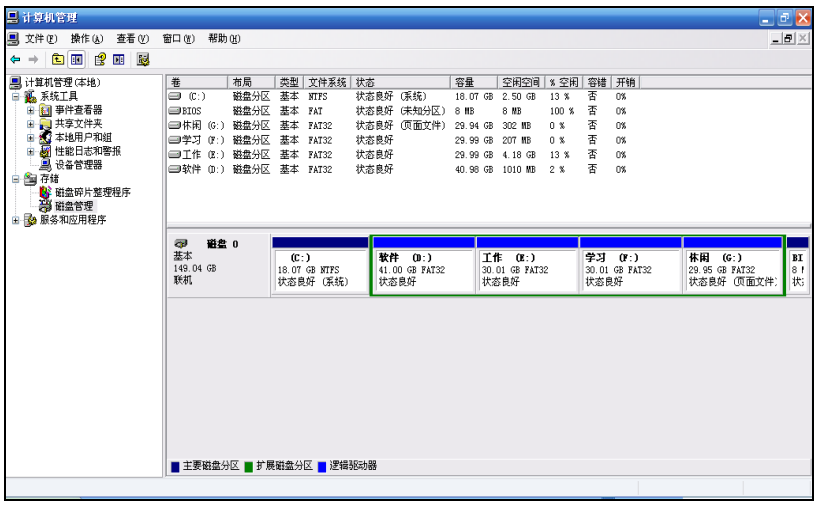


图 18-23　磁盘和磁盘分区的关系

创建名为“ReportDiskPartition.ps1”的脚本，用于获取系统中存在的分区属性。其中将检查变量$args 的值，以判断执行脚本时是否传递参数。如果不存在该变量，则表明在运行脚本时未提供参数。此时脚本会作为本地计算机处理，即传递 localhost 给$args 变量。如果传递问号给脚本，则返回当前脚本的帮助信息，该脚本的代码如下：

例 18-5　获取系统中存在的分区属性

```
    if(!$args)
     {
```

```
    Write-Host -foregroundcolor green `
    'Querying localhost ...'
    $args = 'localhost'
   }
if($args -eq "?")
   { "
      ReportDiskPartition.ps1

        DESCRIPTION:
      This script can take a single argument, computer name.
        It will display drive configuration on either a local
        or a remote computer. You can supply either a ? or a
        name of a local machine.

        EXAMPLE:
        ReportDiskPartition.ps1 remoteComputerName
        reports on disk partition information on a computer named
        remoteComputerName

        The script will also display this help file. This is
        done via the ? argument as seen here.
        ReportDiskPartition.ps1 ?
        "
   }
Get-WmiObject -Class Win32_DiskPartition `
-computer $args
```

其中使用 Get-WmiObject cmdlet 及-class 参数搜索 Win32_DiskPartition WMI 类，并获得磁盘分区的配置信息和值。如果使用$args 参数提供了要查询磁盘分区信息的计算机名，则可使用-computer 参数为 Get-WmiObject 提供计算机名，执行结果如图 18-24 所示。

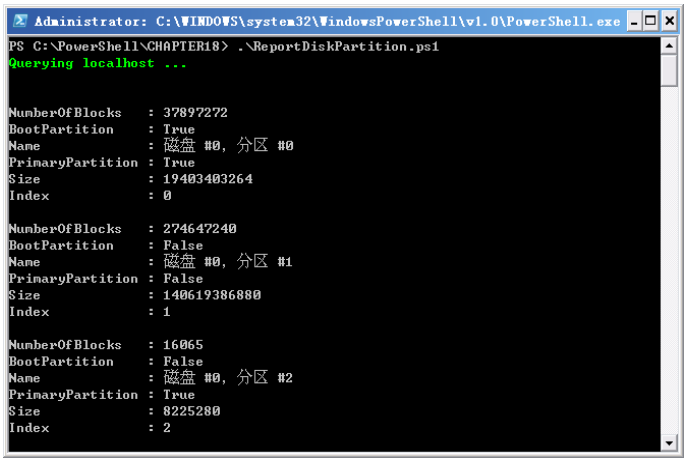


图 18-24　执行结果

（3）匹配磁盘和分区

匹配驱动器和分区之后，还需要相应处理磁盘和分区的脚本，因为有时需要特

定驱动器的分区信息。创建名为“ReportSpecificDiskPartition.ps1”的脚本来获取硬盘特定分区的配置信息。

例 18-6　获取硬盘特定分区的配置信息

```
param($computer="localhost",$disk="磁盘 #0，分区 #0",$help)
if($computer)
 {
  Write-Host -foregroundcolor green `
  "Querying $computer ..."

 }
if($disk)
 {
  Write-Host -foregroundcolor green `
  "Querying $disk for partition information ..."

 }
if($help)
 { "
    ReportSpecificDiskPartition.ps1

      DESCRIPTION:
    This script can take a multiple arguments, computer name,
      drive number and help.
      It will display partition configuration on either a local
      or a remote computer. You can supply either help, drive and
      name of a local or remote machine.

      EXAMPLE:
      ReportSpecificDiskPartition.ps1 -computer remoteComputername
      reports on disk partition on drive 0 on a computer named
      remoteComputerName

      ReportSpecificDiskPartition.ps1 -computer remoteComputername -disk '磁盘 #0，分区 #0'
      reports on disk partition on drive 1 on a computer named
      remoteComputerName


      ReportSpecificDiskPartition.ps1 -help y
      Prints out the help information seen here.

      "
      Exit
 }
Get-WmiObject -Class Win32_DiskPartition `
-computer $computer | Where-Object { $_.name -match $Disk } |
format-list [a-z]*
```

这个脚本的 3 个参数分别是-computer、-disk 和-help，其中-computer 默认为 localhost，即查询本机的分区信息；-disk 指定特定磁盘分区信息，这里将默认值设置为“磁盘#0，分区#0”，即第 1 块硬盘的第 1 个分区；-help 输出脚本的名称、描

述信息及语法范例，输出帮助信息后将会使用 exit 语句退出脚本。该脚本的执行结果如图 18-25 所示。

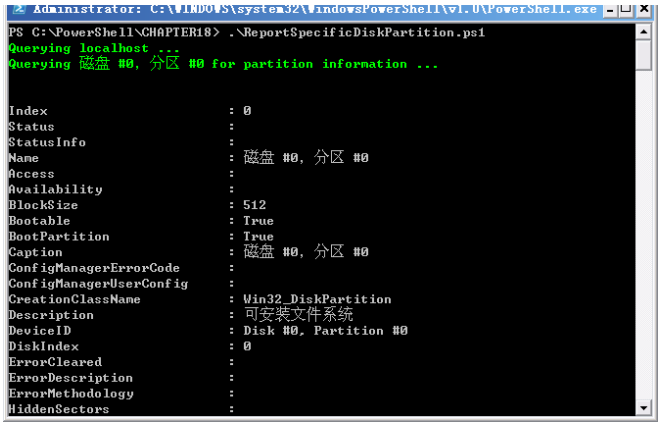


图 18-25　执行结果

（4）处理逻辑磁盘

为了能够获得计算机中有关逻辑磁盘的配置信息，需要使用 Get-WmiObject cmdlet。然后使用-class 参数查询 Win32_LogicalDisk WMI 类，并且可以通过设定-computer 参数查询指定计算机的信息。

所有参数通过$args 变量传递到脚本中，如果该变量不存在，则根据默认值获取第 1 个逻辑磁盘的信息。创建名为“ReportLogicalDiskConfiguration.ps1”的脚本查询系统中存在的逻辑磁盘。

例 18-7　查询系统中存在的逻辑磁盘

```
param($computer="localhost",$disk="C:",$help)
if($computer)
 {
  Write-Host -foregroundcolor green `
  'Querying localhost ...'
 }
if($disk)
 {
  Write-Host -foregroundcolor green `
  'Querying $disk for logical disk info...'
#   $args = 'localhost'
 }
if($help)
 { "
    ReportLogicalDiskConfiguration.ps1

      DESCRIPTION:
    This script can take a single argument, computer name.
      It will display logical disk configuration on either a local
      or a remote computer. You can supply either a ? or a
      name of a local machine.
```

```
        EXAMPLE:
        ReportLogicalDiskConfiguration.ps1 remoteComputerName
        reports on logical disk configuration on a computer named
        remoteComputerName

        The script will also display this help file. This is
        done via the ? argument as seen here.
        ReportLogicalDiskConfiguration.ps1 ?
        "
 }
Get-WmiObject -Class Win32_LogicalDisk `
-computer $computer | Where-Object {$_.deviceID -match $Disk} |
Format-List [a-z]*
```

此脚本的执行结果如图 18-26 所示。

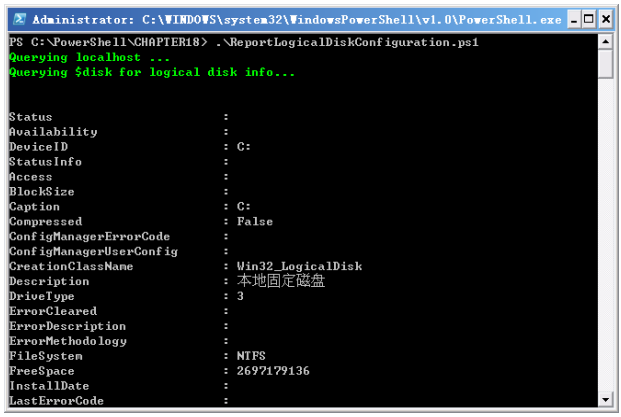


图 18-26　执行结果

（5）重命名计算机

在特定条件下，需要批量重新设置计算机名。创建名为“RenameComputer.ps1”的脚本用于更改计算机名，为此首先通过 Get-WmiObject cmdlet 在指定类为 Win32_Computersystem 的情况下获取对象。通过这个对象的 rename(newname)方法更名计算机。

例 18-8　更改计算机名

```
param(
       $computer="localhost",
       $newName,
       $user = "administrator",
       $password,
       $help
      )
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: RenameComputer.ps1
```

```
    Renames a local or remote machine.
    PARAMETERS:
    -computer Specifies the name of the computer upon which to run the script
    -newname   new name of the computer
    -user      user credentials
    -password password of the user
    -help      prints help file
    SYNTAX:
    RenameComputer.ps1 -computer BeijingServer -newname ShanghaiServer
    Renames a computer named BeijingServer to ShanghaiServer
    RenameComputer.ps1 -computer BeijingServer -newname ShanghaiServer
    -user WebServer\admin -password MyPassword
    Renames a computer named BeijingServer to ShanghaiServer. Uses
    the credentials of the WebServer admin, with password of MyPassword
    RenameComputer.ps1
    Generates an error. Must supply new name for computer
    RenameComputer.ps1 -help ?
    Displays the help topic for the script
    "@
    $helpText
    exit
    }
    if($help){ "Obtaining help ..." ; funhelp }
    if(!$newname){Write-Host "Need Some Parameters,You can get help by
'RenameComputer.ps1 -help ?' ";exit}
    if($computer -ne "localhost")
     {
      $objWMI = Get-WmiObject -Class Win32_Computersystem `
      -computername $computer -credential $user
      $objWMI.rename($newName)
     }
    ELSE
     {
      $objWMI = Get-WmiObject -Class Win32_Computersystem `
      -computername $computer
      $objWMI.rename($newName)
     }
```

其中将要输出的帮助内容拼接为字符串，帮助功能的实现是将帮助字符串封装到函数中，在需要时直接调用即可。该脚本可以接受的 5 个参数分别是$computer、$newName、$user、$password 和$help，其中$computer 和$user 的默认值分别为 localhost 和 administrator。设置默认值的目的在于当在本机执行脚本时，用户未传递主机名和用户名即可仅传递新主机名更改本机主机名；$password 是在更名远程主机时需要的密码。图 18-27 所示为用户在执行更名操作之前输入脚本名时获取的提示信息，其后为按照查看帮助后的标准格式提供参数的更名效果。

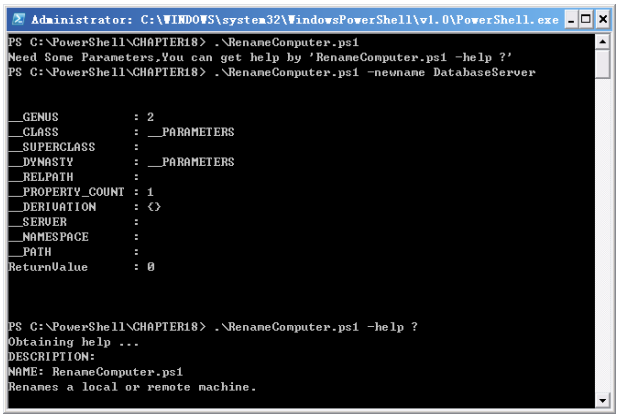


图 18-27　更改当前主机名及其结果

（6）关闭或重启远程计算机

在执行更名主机或添加域操作后，为了使设置生效需要重启计算机。为此需要使用 Win32_OperatingSystem WMI 类的 shutdown()和 reboot()方法，要执行的操作由向脚本传递的参数-a 确定，值为 s 则关机；为 r 则重启。为了顺利地关机或重启所用账户必须具有相应的权限，将 EnablePrivileges 的属性设置为$true。

需要注意的是如果执行关机和重启操作的主机是本机，则需要两次定义 Get-WmiObject，分别为出示凭据和不需要使用凭据的情况。如果操作的主机不是本机，则需要使用备用凭据。

```
param(
      $computer="localhost",
      $user = "administrator",
      $password,
      $a,
      $help
 )
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: ShutdownRebootComputer.ps1
Shutdown or reboot a local or remote machine.
ARAMETERS:
-computer Specifies the name of the computer upon which to run the script
-user      user credentials
-password password of the user
-a(ction) action to perform < s(hutdown), r(eboot) >
-help      prints help file
SYNTAX:
ShutdownRebootComputer.ps1 -computer WebServer -a s
Shutdown a remote computer named WebServer
ShutdownRebootComputer.ps1 -computer WebServer -a r
```

```
-user WebServer\admin -password MyPassword
Reboots a computer named WebServer. Uses the credentials
of the WebServer admin, with password of MyPassword
ShutdownRebootComputer.ps1
Displays message pointing to help
ShutdownRebootComputer.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
if($help){ "Obtaining help ..." ; funhelp }
switch($a)
{
 "s" {
       if($computer -ne "localhost")
       {
        $objWMI = Get-WmiObject -Class Win32_operatingsystem `
        -computername $computer -credential $user
        $objWMI.psbase.Scope.Options.EnablePrivileges = $true
        $objWMI.shutdown()
       }
       ELSE
       {
        $objWMI = Get-WmiObject -Class Win32_operatingsystem `
        -computername $computer
        $objWMI.psbase.Scope.Options.EnablePrivileges = $true
        $objWMI.shutdown()
       }
      }
 "r" {
       if($computer -ne "localhost")
       {
        $objWMI = Get-WmiObject -Class Win32_operatingsystem `
        -computername $computer -credential $user
        $objWMI.psbase.Scope.Options.EnablePrivileges = $true
        $objWMI.reboot()
       }
       ELSE
       {
        $objWMI = Get-WmiObject -Class Win32_operatingsystem `
        -computername $computer
        $objWMI.psbase.Scope.Options.EnablePrivileges = $true
        $objWMI.reboot()
       }
      }
 DEFAULT { "You must supply an action. Try this"
           "ShutdownRebootComputer.ps1 -help ?" }
}
```

执行此脚本调用命令.\ShutdownRebootComputer.ps1 –a s和.\ShutdownRebootComputer.ps1 –a r，分别对应当前系统的关闭和重启。关机和重启操作在本地计算机上可以直接执行，而远程计算机需要指定相应的参数，其中-computer 指定主机名；-user 指定用户；-password 指定用户密码。

为了避免误运行该脚本关闭或重启计算机，将默认操作设置为显示帮助字符串，并要求用户在提供-help 参数的情况下才能运行该脚本的帮助信息。

18.3.2　监控磁盘空间

随着时间的增长，系统中的可用空间会越来越少。系统管理员需要清理磁盘，前提是了解系统磁盘空间，为此需要追踪一段时间内的磁盘空间的使用情况。创建名为“QueryOldFiles.ps1”的脚本连接到特定文件夹，并为其中所有超过 30 天未访问过的文件生成列表，在获得过期文件后可以根据实际情况处理。

例 18-9　获得过期文件并生成列表

```
function FunWithLine ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}
$folder = "c:\Windows"
$date = Get-Date
$limit = 30
Get-ChildItem -Path $folder -force |
foreach-object `
{
 $newDate=($_.LastAccessTime).adddays($limit)
 $limitDate = New-TimeSpan -start $date -end $newDate
 if ($limitDate -le 0)
  {
   $xfiles += @{ $_.name = $_.lastAccessTime }
  }

}
Write-Host "There are $($xfiles.count) files from $folder greater than $limit days old."
FunWithLine("The expired files are listed below:")
$xfiles
```

其中首先定义了 FunWithLine 函数获得传递给函数的字符串的长度，并使用 Write-Host cmdlet 创建用黄色显示带下画线的字符。然后初始化 3 个变量，其中是$folder 保存要检查过期文件的文件夹的路径；$data 保存当前日期的 datetime（日期时间）对象。当前日期可以通过 Get-Date cmdlet 获取，$limit 变量值用于判断允许文件存在的最长寿命，可以据此决定是否需要将其报告为过期文件。

在初始化 3 个变量之后使用 Get-Children cmdlet 获得要检查的文件列表，将$folder 变量中包括的字符串提供给 Get-Children cmdlet 的-path 参数。然后使用-force 参数返回隐藏文件，并将生成的 fileinfo 对象采用管道传递给 ForEach-Object cmdlet。

在 ForEach-Object 代码块中创建了新的 datetime 对象并将其保存在$newDate 变量中，为了在文件现有最后访问时间点的基础上增加天数$limit，使用了 addDays()方法。datetime 对象通过查询管道中当前 fileinfo 对象的 LastAccessTime 属性获得。

使用 New-TimeSpan cmdlet 创建代表文件的有效存活时间段，其中该对象为当前时间$date 变量提供-start 参数。将$newDate 变量提供给-end 参数，表示文件最后访问日期要求的过期时间$limit，最后在$limitdate 变量中保存获得的 timespan 对象。当$limitdate 值小于等于 0 时，则将当前对象输出给$xfiles 变量，表示文件的最后访问日期已经超过了$limit 变量中指定的过期时间。图 18-28 所示即此脚本在 Windows 文件夹中的执行结果，需要提示的是执行此脚本会因为 Windows 文件夹中文件的过期时间不同而显示不同的结果，但其原理一致。

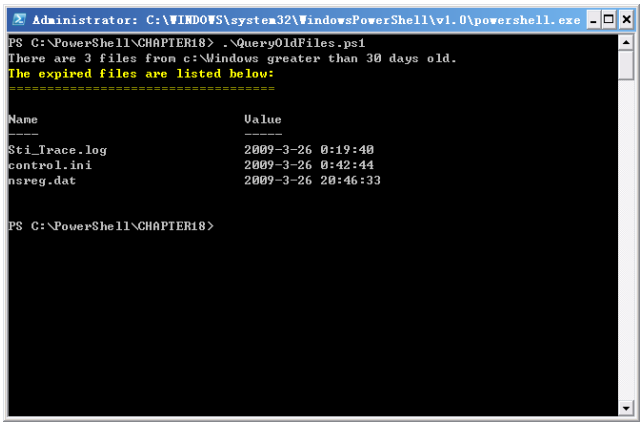


图 18-28　执行结果

18.3.3　用户管理

系统管理员的重要任务之一是管理用户，既允许用户做执行相应的操作，又要防止因权限设置不合理而潜在的用户越权操作。如在虚拟主机中设置的用户主目录的权限有问题，其他用户可以采用..反溯到上级目录。进而访问其他用户的文件，直接造成这些文件的泄露，这个问题的解决方法是将每个用户的主目录权限设置为仅有管理员和用户本人可以访问，而拒绝其他人的访问。各用户的文件只是独立在自己的权限范围内，无法访问他人的文件，他人也无法访问自己的文件。

（1）创建本地用户

创建本地用户的方法通常使用 net user 和 ADSI 命令，也可以使用如图 18-29 所

示的“新用户”对话框。

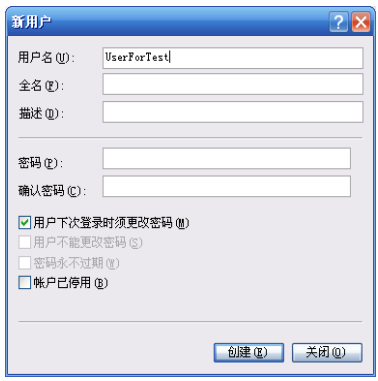


图 18-29　“新用户”对话框

在 cmd 下可以直接使用 net user 命令创建用户。在 PowerShell 中使用 ADSI 方便而有效地创建本地用户和组，也可以使用 WinNT ADSI Providernet user 命令。本地账户没有域账户的多个属性，因而创建过程很简单。例 18-10 可创建名为“CreateLocalUser.ps1”的脚本。

例 18-10　创建域账户

```
param($computer="localhost", $group, $help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: CreateLocalGroup.ps1
Creates a local group on either a local or remote machine.
PARAMETERS:
-computer Specifies the name of the computer upon which to run the script
-group     Name of group to create
-help      prints help file
SYNTAX:
CreateLocalGroup.ps1
Generates an error. You must supply a group name
CreateLocalGroup.ps1 -computer WebServer -group MyGroup
Creates a local group called MyGroup on a computer named WebServer
CreateLocalGroup.ps1 -group Mygroup
Creates a local group called MyGroup on local computer
CreateLocalGroup.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
if($help){ "Obtaining help ..." ; funhelp }
if(!$group)
      {
       $(Throw 'A value for $group is required.
       Try this: CreateLocalGroup.ps1 -help ?')
```

```
    }

$OBJou = [ADSI]"WinNT://$computer"
$objUser = $objOU.Create("Group", $group)
$objUser.SetInfo()
$objUser.description = "Test Group"
$objUser.SetInfo()
```

其中首先使用 param 语句定义了 4 个参数，即-computer、-user、-password 和-help，随后定义了 funhelp 函数输出帮助信息。如果$help 变量存在，则显示一条信息提示正在获取帮助文件，然后调用 funhelp 函数。接下来检查输入内容中的-user 和-password 参数是否被赋值，如果值不存在，则直接调用 throw 语句产生错误信息，并停止执行后续代码。

在用户名和密码均存在的前提下，脚本调用[ADSI]类型约束连接到本地计算机的账户数据库，并使用 Create()方法利用$user 变量中提供的用户名创建账户。随后调用 SetPasswod()方法设置密码，并调用 SetInfo()方法将其写入数据库。最后设置 Description 属性，并再次调用 SetInfo()。

需要强调的是这个脚本中使用的 SetInfo()方法仅在 Windows Vista 和 Windows 2008 以后版本的 WMI 模型中才出现，在 Windows XP 下安装的 PowerShell 中执行则报错。使用这个脚本创建用户的方法如下：

```
.\CreateLocalUser.ps1 –computer localhost -user UserForTest -password Passw0rd#!
```

创建的用户能够在管理工具中看到。

（2）创建用户组

为了有效地控制访问本地资源（如共享文件夹、扫描仪或打印机），需要在 Windows Vista 或 Windows 2008 系统中创建本地用户组。这些本地用户组如图 18-30 所示。

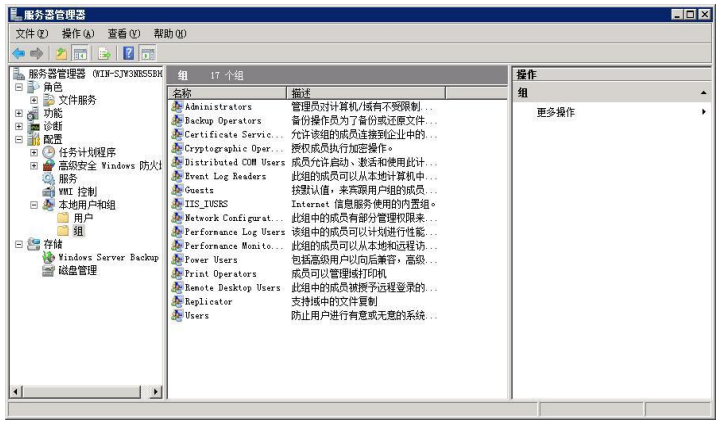


图 18-30　本地用户组

本地用户组同样使用工作组设置，与新建账户工具类似。计算机管理控制台中

也提供了“新建组”对话框，如图 18-31 所示。

图 18-31　“计算机管理”控制台中的“新建组”对话框

为了创建本地用户组，创建名为“CreateLocalGroup.ps1”的脚本，其代码如下：

例 18-11　创建本地用户组

```
param($computer="localhost", $group, $help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: CreateLocalGroup.ps1
Creates a local group on either a local or remote machine.
PARAMETERS:
-computer Specifies the name of the computer upon which to run the script
-group     Name of group to create
-help      prints help file
SYNTAX:
CreateLocalGroup.ps1
Generates an error. You must supply a group name
CreateLocalGroup.ps1 -computer WebServer -group MyGroup
Creates a local group called MyGroup on a computer named WebServer
CreateLocalGroup.ps1 -group Mygroup
Creates a local group called MyGroup on local computer
CreateLocalGroup.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
if($help){ "Obtaining help ..." ; funhelp }
if(!$group)
      {
       $(Throw 'A value for $group is required.
       Try this: CreateLocalGroup.ps1 -help ?')
  }
```

```
$OBJou = [ADSI]"WinNT://$computer"
$objUser = $objOU.Create("Group", $group)
$objUser.SetInfo()
$objUser.description = "Test Group"
$objUser.SetInfo()
```

其中首先使用 param 语句定义了 3 个参数，即-computer、-group 及-help，并将-computer 参数的默认值设置为 localhost。然后定义 funhelp 函数输出帮助信息，当用户提供-help 参数时将输出帮助信息。

在运行脚本时需要提供新建组的名称，如果未提供$group 变量，则显示通过 throw 语句产生的错误信息。创建用户组不需要提供密码，使用该脚本创建用户组的方法如下：

```
.\CreateLocalGroup.ps1 -group NewGroups
```

创建的用户组在计算机管理控制台中的本地组信息中显示，该脚本也使用了仅适用于 Windows Vista 和 Windows 2008 以上系统的 SetInfo()方法。

（3）禁用或启用用户账号

本地用户账号主要用于访问本地资源或本地服务的账号，系统管理员可以禁用或启用特定账户来控制账户的可用性。

创建名为“EnableDisableUser.ps1”的脚本禁用或启用特定账户的权限。

例 18-12　禁用或启用特定账户的权限

```
param($computer="localhost", $a, $user, $password, $help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: EnableDisableUser.ps1
Enables or Disables a local user on either a local or remote machine.
PARAMETERS:
-computer Specifies the name of the computer upon which to run the script
-a(ction) Action to perform < e(nable) d(isable) >
-user       Name of user to modify
-help       prints help file
SYNTAX:
EnableDisableUser.ps1
Generates an error. You must supply a user name
EnableDisableUser.ps1 -computer WebServer -user myUser
-password Passw0rd^&! -a e
Enables a local user called myUser on a computer named WebServer
with a password of Passw0rd^&!
EnableDisableUser.ps1 -user myUser -a d
Disables a local user called myUser on the local machine
EnableDisableUser.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
```

```
}
$EnableUser = 512 # ADS_USER_FLAG_ENUM enumeration value from SDK
$DisableUser = 2   # ADS_USER_FLAG_ENUM enumeration value from SDK
if($help){ "Obtaining help ..." ; funhelp }
if(!$user)
       {
        $(Throw 'A value for $user is required.
        Try this: EnableDisableUser.ps1 -help ?')
   }
$ObjUser = [ADSI]"WinNT://$computer/$user"
switch($a)
{
 "e" {
       if(!$password)
     {
         $(Throw 'a value for $password is required.
         Try this: EnableDisableUser.ps1 -help ?')
     }
       $objUser.setpassword($password)
       $objUser.description = "Enabled Account"
       $objUser.userflags = $EnableUser
       $objUser.setinfo()
 }
 "d" {
       $objUser.description = "Disabled Account"
       $objUser.userflags = $DisableUser
       $objUser.setinfo()
 }
 DEFAULT
        {
       "You must supply a value for the action.
       Try this: EnableDisableUser.ps1 -help ?"
      }
}
```

使用该脚本可以启用或禁用特定账户，并更改特定账号的密码。在脚本中首先使用 param 语句定义了 5 个参数，其中-compute 指定执行该脚本的主机，默认为本机 localhost；-a 指定运行脚本执行的操作；-user 和-password 指定密码；-help 调用 funhelp 函数显示帮助的内容。在 funhelp 函数之后声明了两个变量，其中包括 ADS_USER_FLAG_ENUM 枚举值，相关具体的内容可以参考 Windows SDK，这些值可以用于启动或禁用用户账号。

需要强调的是虽然 ADS_USER_FLAG_ENUM 枚举值在 Windows SDK 中有详细说明，但是并没有介绍在 PowerShell 中的使用。由于无法获得 PowerShell 中 IadsUser 的直接支持，所以在 VBScript 中用于禁用账户的 AccountDisabled 布尔属性在 PowerShell 中无法继续使用。这样之前用于禁用账户的 VBScript 脚本不能移植到 PowerShell 中，而 EnableDisableUser.ps1 脚本并不能由 VBScript 转换而来。

定义这两个变量后需要测试 help 参数以确定是否显示帮助信息，在这个脚本中必须输入参数-user，即禁用或启用的用户名。如果该参数为空，系统将会抛出错误

并停止后续语句的执行。检测到用户名后，使用[ADSI]类型约束符及 WinNT Active Directory 服务接口（ADSI）Provider 连接到本地计算机的 SAM 账户数据库并获取用户对象。

获取用户对象之后使用 switch 语句判断$a 变量的值，该变量用于指定脚本中执行的相应操作。如果要启用账户，那么需要为该账户设置密码，即为参数-a 提供 e（enable，启用）值。当-password 参数未接收到密码时抛出异常提示用户查看帮助信息。如果密码存在，则调用 SetPasswod 方法为用户设置密码。并将用户 Description（描述）属性修改为“Enable Account”，使用 UserFlags 属性值启用账户，最后调用 SetInfo()方法将更改提交到 SAM 账户数据库。启用账户操作命令的标准格式如下：

```
.\EnableDisableUser.ps1 -user UserForTest -password Passw0rd#! -a e
```

要禁用账户，为参数-a 传递 d（disable，禁用）值。并为 UserFlags 参数设置适当值，调用 SetInfo()方法将更改提交给 SAM 账户数据库。在这里为了使演示效果更为明显，在调用 SetInfo()方法之前将用户的 Description（描述）属性改为 disable Account。禁用账户操作命令的标准格式如下：

```
.\EnableDisableUser.ps1 -user UserForTest -a d
```

如果用户未指定-a 的参数为 e 或 d，则脚本执行默认操作，在本例中会输出字符串提示用户查看帮助文件。启用或禁用用户的执行结果如图 18-32 所示。

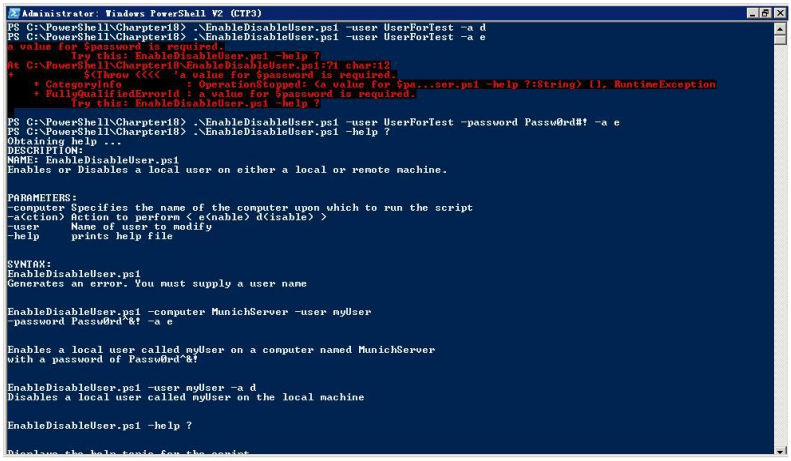


图 18-32　执行结果

18.3.4　设置桌面选项

Windows 系统需要针对图形界面设置有关选项，如屏幕保护程序、桌面及电源设置等。尽管有很多用户通过组策略方式配置这些选项，但是对于尚未部署活动目录（Active Directory）的企业还是在使用默认的组策略对象，为需要通过 PowerShell 实现这些功能。

（1）设置屏幕保护程序

设置桌面选项时首先要注意 PC 是否配置了屏幕保护程序，在处理屏幕保护程序时需要考虑系统的性能。如在服务器上不必使用复杂的 3D 场景作为屏幕保护程序，因为这样既耗费服务器的资源，又没有多大意义；此外需要注意安全问题，因为有些屏幕保护程序会联系外部服务器更新配置并报告用户操作习惯。如果更新过程被恶意操控，将会给服务器的安全带来很大威胁。如果为负载多个不同用户网站的 Web 服务器设置的权限不合适，恶意用户可以用更名后的木马程序替换原有屏幕保护程序。从而借助屏幕保护程序在达到启动时间后启动木马，并且逐步提升权限控制整台服务器。

创建名为“AuditScreenSaver.ps1”的脚本检查特定计算机中的屏幕保护程序是否显示适当内容，同时检测其名称、等待时间，以及是否安全。其代码如下：

例 18-13　检查屏幕保护程序

```
param($computer="localhost", $help)
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: AuditScreenSaver.ps1
Prints screensaver config on a local or remote machine.
PARAMETERS:
-computerName Specifies the name of the computer upon which to run the script
-help           prints help file
SYNTAX:
AuditScreenSaver.ps1 -computer WebServer
Lists screensaver configuration on a computer named WebServer
AuditScreenSaver.ps1
Lists screensaver configuration on local computer
AuditScreenSaver.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
if($help){funline("Obtaining help ...") ; funhelp }
$username = (get-wmiobject -class win32_computersystem `
  -computername $computer).username
$index=$username.indexof("\")
$username=$username.substring($index+1)
$screensaver = Get-WmiObject -Class win32_desktop `
```

```
  -computername $computer -filter "name like `"%$($username)`"" |
Select-Object -Property screen*, name
funline("Screen saver configuration for $($screensaver.name)")
if($screensaver.ScreenSaverActive -eq "true")
 {
  Write-Host "The screensaver is: $($screensaver.screensaverExecutable)"
  Write-Host "Secure Screensaver: $($screensaver.ScreenSaverSecure)"
  Write-Host "Screensaver timeout: $($screensaver.ScreenSaverTimeout)"
 }
ELSE
 { Write-Host "$($screensaver.name) does not have a screen saver"}
```

该脚本首先使用 param 语句定义了两个输入参数，其中$computer 指定运行该脚本的计算机，默认值为 localhost，即本机；$help 指定是否显示帮助信息。然后定义 funline 函数，用于为传递的字符串添加下画线，为用户提供更好的视觉反馈。该函数会判断传递的字符串的长度，它决定了需要使用多少个等号（=）连接，通过这种方式组成的字符串的长度可以等于输出字符串的长度。

接下来定义了一个 funhelp 函数用于在用户未输入相关参数或输入了错误参数时提示用户按照脚本描述和语法输入正确的参数。这个脚本可以远程运行，但是需要获取当前登录的用户名称。为此可以使用 Get-WmiObject cmdlet 并通过 Win32_ComputerSystemWMII 类获取 UserName 属性，此处使用-computername 参数并提供$computer 变量中的数值。

随后使用 Get-WmiObject cmdlet 查询 Win32_Desktop WMI 类，这时可以使用 -computername 参数使脚本以远程计算机为目标运行。如果如图 18-33 所示关闭屏幕保护程序，而且尚未重启系统，则无法更新该屏幕保护程序的属性值，因为当前配置的注册表键值只有在系统重启后才会生效。

图 18-33　关闭屏幕保护程序

如果 ScreenSaverActive 的属性为 True，则输出可执行文件的路径，在其中可以看到当前屏幕保护程序设置的等待时间。这个等待时间以秒为单位，因此默认的 10 分钟显示为 600。

此脚本的执行结果如图 18-34 所示。

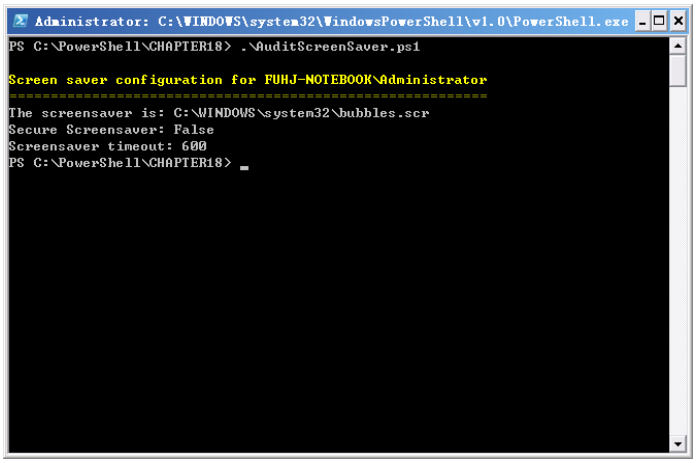


图 18-34　执行结果

（2）审核安全的屏幕保护程序

屏幕保护程序也有潜在的安全威胁，如果用户主机中存在大量的用户，则系统管理员逐个查看用户的屏幕保护程序设置是否安全，以及设置是否按照要求将会是个很繁重的任务。创建名为“AuditScreenSaverWriteToAccess.ps1”脚本用于获取和审核主机中所有用户在系统中的屏幕保护程序设置，并将结果写入名为 ConfigurationMaintenance.mdb 的 Access 数据库文件中。创建一个新表 screensaver 用于保存当前信息，其结构如 18-35 所示。

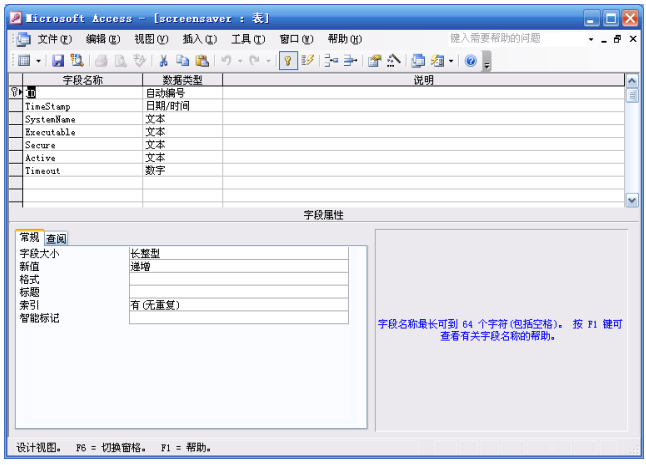


图 18-35　screensaver 表的结构

创建一个 screensaver 表，如图 18-36 所示。其中列出 screensaver 表中的所有信息，并计算使用屏幕保护程序用户的比例，同时指出使用的屏幕保护程序是否安全。

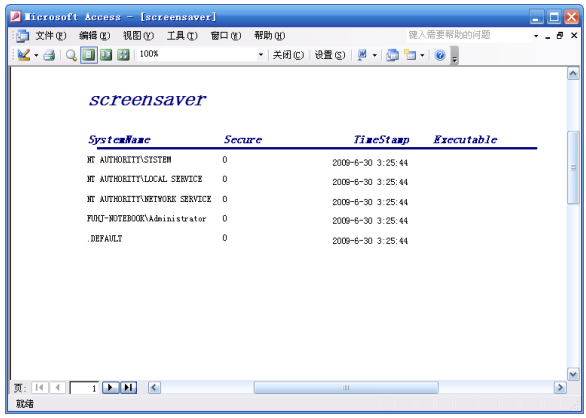


图 18-36　screensaver 表

例 18-14　创建一个 screensaver 表

```
param($computer="localhost", $help)
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: AuditScreenSaverWriteToAccess.ps1
writes secure screensaver config of a local or remote machine,
to an access database
PARAMETERS:
-computerName Specifies the name of the computer upon which to run the script
-help          prints help file
SYNTAX:
AuditScreenSaverWriteToAccess.ps1 -computer WebServer
Writes secure screensaver configuration of a computer named WebServer
to an access database
AuditScreenSaverWriteToAccess.ps1
Writes secure screensaver configuration of local computer to an
access database
AuditScreenSaverWriteToAccess.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
if($help){funline("Obtaining help ...") ; funhelp }
$adOpenStatic = $adLockOptimistic = 3
$strDB = "C:\PowerShell\CHAPTER18\Configurationmaintenance1.mdb"
$strTable = "screensaver"
$objConnection = New-Object -ComObject ADODB.Connection
```

```
$objRecordSet = new-object -ComObject ADODB.Recordset
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
  Data Source= $strDB")
$objRecordSet.Open("SELECT * FROM $strTable", `
$objConnection, $adOpenStatic, $adLockOptimistic)
write-host -foreGroundColor yellow "Obtaining screen saver info ..."
$aryscreensaver = Get-WmiObject -Class win32_desktop `
  -computername $computer `
  -Property name, screensaversecure, screensavertimeout, `
  __server, ScreenSaverActive
foreach( $screensaver in $aryScreensaver)
{
   $objRecordSet.AddNew()
   $objRecordSet.Fields.item("TimeStamp") = Get-Date
   $objRecordSet.Fields.item("SystemName") = $($screensaver.name)
   $objRecordSet.Fields.item("Executable") = $($screensaver.screensaverExecutable)
   $objRecordSet.Fields.item("Secure") = $($screensaver.ScreenSaverSecure)
   $objRecordSet.Fields.item("Active") = $($screensaver.ScreenSaverActive)
   $objRecordSet.Fields.item("Timeout") = $($screensaver.ScreenSaverTimeout)
   $objRecordSet.Update()
   write-host -foregroundColor yellow "/\" -noNewLine
}
$objRecordSet.Close()
$objConnection.Close()
```

其中定义了控制打开 Access 数据库连接方式的$adOpenStatic 和$adLockOptimistic 两个变量并均赋值 3，该值根据 Windows SDK 定义。然后将 Configurationmaintenance.mdb 数据库的路径赋给$strDB 变量，并使用$strTable 变量保存将要写入内容的表名，即该脚本中使用的 screensaver 表。为了使用脚本操作 Access 数据库，必须创建连接到数据库并更新表中信息的 Connection 和 RecordSet 对象。接下来打开与数据库的连接，需要提供 Provider 的名称，即 Microsoft.Jet.OLEDB4.0 Provider，然后使用 ADODB.Connection 对象的 Open 方法打开$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0;Data Source= $strDB")。打开数据库连接后，使用 RescordSet 对象的 Open 方法打开记录集，并使用 Write-Host cmdlet 输出信息提示用户正在收集系统屏幕保护程序的信息。

为了通过 WMI 类收集屏幕保护程序的信息，使用 Get-WmiObject cmdlet 并选择 Win32_Desktop WMI 类。使用-computer 参数通过$computer 变量提供目标计算机的名称，并通过-property 参数选择所需属性。然后使用 foreach 语句遍历上面通过 WMI 获取的屏幕保护程序，使用 RecordSet 为每个记录添加一条新记录，并且设置记录中每个字段的内容，最后使用 RecordSet 对象的 Update()方法更新所有字段的内容。在添加每一条记录的过程中在控制台输出“/\”符号显示进度，foreach 循环遍历所有记录后需要牢记使用各自的 Close()方法关闭数据库的记录集 RecordSet 对象和连接 Connection 对象。

执行脚本之后打开数据库 ConfigurationMaintenance.mdb 文件，可以看到如图 18-36 所示的内容。

（3）管理桌面电源设置

便携式计算机的供电是个很重要的问题，因其直接关系到便携式设备的可移动性。Windows 有多个组件与电源管理的配置策略相关，在 Windows Server 2008 中的电源策略设置界面如图 18-37 所示。

图 18-37　Windows Server 2008 中的电源策略设置界面

创建名为“ReportPowerConfig.ps1”的脚本，根据用户提供的如下参数提供相应的电源配置信息。

- a：当前主机中活动的电源设置。
- l：当前主机中的所有电源配置。
- q：当前主机中的所有可用休眠状态。
- w：当前主机中的上次唤醒事件。
- d：当前主机中的所有设备。
- dv：当前主机中的所有设备的详细信息。
- dwa：当前主机中已配置且可唤醒当前主机的设备。
- dwp：当前主机中所有配置为可从睡眠中唤醒计算机的设备。

例 18-15　提供电源配置信息

```
param($a="a", $help)
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: ReportPowerConfig.ps1
```

```
Prints power config on a local machine.
PARAMETERS:
-a(ction) action to perform <a(ctive scheme), l(ist),
           q(uery), d(evice), dv(evice verbose),
        dwa(evice wake armed), dwp(evice wake programable)>
-help      prints help file
SYNTAX:
ReportPowerConfig.ps1
Lists power configuration on local computer
ReportPowerConfig.ps1 -a a
Lists active power configuration on local computer
ReportPowerConfig.ps1 -a l
Lists all power configuration on local computer
ReportPowerConfig.ps1 -a q
Lists all available sleep states on local computer
ReportPowerConfig.ps1 -a w
Lists last wake event on local computer
ReportPowerConfig.ps1 -a d
Lists all devices on local computer
ReportPowerConfig.ps1 -a dv
Lists all devices on local computer - verbose
ReportPowerConfig.ps1 -a dwa
Lists devices configured to wake the local computer
ReportPowerConfig.ps1 -a dwp
Lists devices that are user confiurable to wake the
computer from sleep on local computer
ReportPowerConfig.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
if($help){funline("Obtaining help ...") ; funhelp }
$computer = (New-Object -ComObject WScript.Network).computername
funline("Power configuration on: $($computer)")
switch($a)
{
 "a"    { POWERCFG -GETACTIVESCHEME ; "`r"}
 "l"    { powercfg -LIST }
 "q"    { powercfg -AVAILABLESLEEPSTATES }
 "w"    { powercfg -lastwake }
 "d"    { powercfg -devicequery all_devices }
 "dv"   { powercfg -devicequery all_devices_verbose }
 "dwa" { powercfg -devicequery wake_armed }
 "dwp" { powercfg -devicequery wake_programmable }
 }
```

在脚本中首先使用 param 语句定义了两个命令行参数-a 和-help，-a 指定脚本执行的操作，默认值为 a；-help 指定显示帮助信息，包括描述、参数及语法范例。要强调的是这个脚本无法在远程执行。

获取计算机名称时可以使用 WScript.Network COM 对象，并使用 New-Object cmdlet 创建该对象，然后提供-comobject 参数。选择 ComputerName 属性，计算机名自动保存在$computer 变量中。该脚本中的大部分的逻辑控制在 switch 语句中完

成，这些语句通过判断命令行中的$a 变量值选择相应的分支。如果值为 a，则使用 Powercfg 工具获取当前电源计划。执行结果如图 18-38 和图 18-39 所示，可以看到当前计算机可以由网卡远程唤醒。需要强调的是由于 Windows XP 和 Windows Server 2008 的 powercfg 工具的工作环境不同，所以将-a 参数的 a 或 q 选项传递给脚本使将会抛出“参数无效，输入"/?"得到帮助”的提示信息。

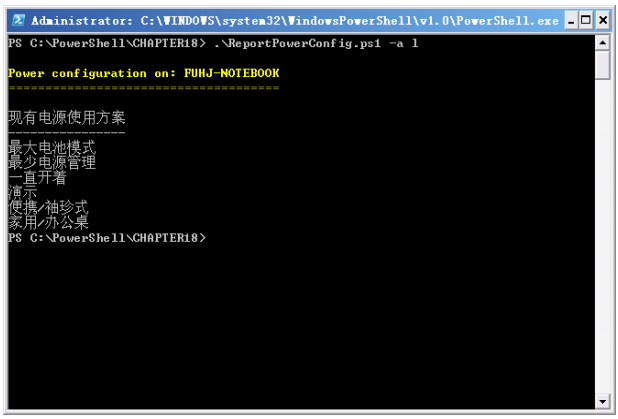


图 18-38　列出当前系统中可用的电源计划

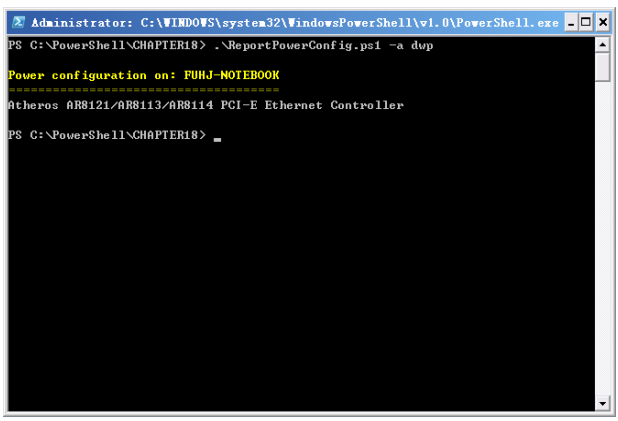


图 18-39　获取可从睡眠中唤醒计算机的设备

（4）修改电源计划

Windows Vista 和 Windows Server 2008 的电源计划有大量改进，可以针对使用电池或交流电供电的情况分别设置。如果当前计算机正在使用电池，那么续航时间是个需要关心的问题。而在某些情况下，计算机的性能才是最重要的。例如，如果电力会在几分钟之后恢复，则不必为延长电池使用时间而降低性能。在不同的电源计划下显示器、磁盘及 CPU 的电力消耗也不同，在 Windows Server 2008 系统中可以创建自定义的电源计划，如图 18-40 所示。

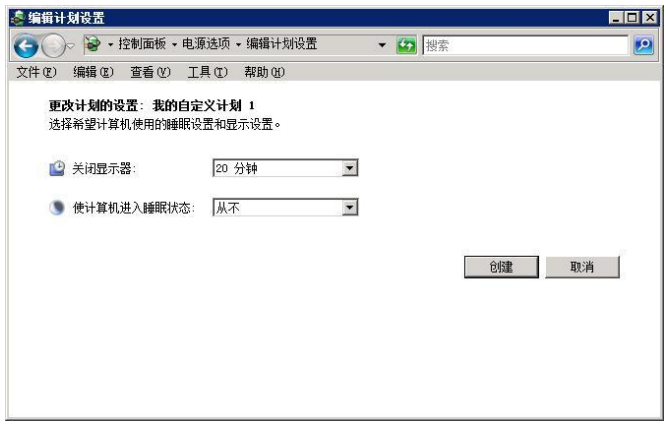


图 18-40　创建自定义的电源计划

例 18-16 创建名为“SetPowerConfig.ps1”的脚本用于设置电源计划。

例 18-16　设置电源计划

```
param($c, $t, $q, $help)
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: SetPowerConfig.ps1
Sets power config on a local machine.
PARAMETERS:
-c(hange)    <mp,mb,dp,db,sp,sb,hp,hb>
-q(uery)     detailed query of current power plan
-t(ime out) time out value for change. Required when using -c to change a value
-help        prints help file
SYNTAX:
SetPowerConfig.ps1
Displays error message. Must supply a parameter
SetPowerConfig.ps1 -c mp -t 10
Sets time out value of monitor when on power to10 minutes
SetPowerConfig.ps1 -c mb -t 5
Sets time out value of monitor when on battery to 5 minutes
SetPowerConfig.ps1 -c dp -t 15
Sets time out value of disk when on power to 15 minutes
SetPowerConfig.ps1 -c db -t 7
Sets time out value of disk when on battery to 7 minutes
SetPowerConfig.ps1 -c sp -t 30
Sets time out value of standby when on power to 30 minutes
SetPowerConfig.ps1 -c sb -t 10
```

```
Sets time out value of standby when on battery to 10 minutes
SetPowerConfig.ps1 -c hp -t 45
Sets time out value of hibernate when on power to 45 minutes
SetPowerConfig.ps1 -c hb -t 15
Sets time out value of hibernate when on battery to 15 minutes
SetPowerConfig.ps1 -q c
Lists detailed configuration settings of the current power scheme
SetPowerConfig.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
if($help){funline("Obtaining help ...") ; funhelp }
$computer = (New-Object -ComObject WScript.Network).computername
if($q)
{
 funline("Power configuration on: $($computer)")
 powercfg -query
 exit
}
if($c -and !$t)
       {
        $(Throw 'A value for $t is required.
        Try this: SetPowerConfig.ps1 -help ?')
       }
switch($c)
{
 "mp" { powercfg -CHANGE -monitor-timeout-ac $t }
 "mb" { powercfg -CHANGE -monitor-timeout-dc $t }
 "dp" { powercfg -CHANGE -disk-timeout-ac $t}
 "db" { powercfg -CHANGE -disk-timeout-dc $t }
 "sp" { powercfg -CHANGE -standby-timeout-ac $t }
 "sb" { powercfg -CHANGE -standby-timeout-dc $t }
 "hp" { powercfg -CHANGE -hibernate-timeout-ac $t }
 "hb" { powercfg -CHANGE -hibernate-timeout-dc $t }
 DEFAULT {
            "$c is not allowed. Try the following:
            SetPowerConfig.ps1 -help ?"
          }
}
```

该脚本首先使用 param 语句定义了 4 个参数，即-c、-t、-q 和-help。其中-q 指定查询，-help 指定输出帮助。-c 和-t 必须同时使用，因为-t 值指定修改电源计划的参数超时值，如果该值为空，脚本执行将出错。该脚本使用 WScript.Network 对象获得本机的计算机名，因此可以使用 New-Object cmdlet 配合-comobject 参数。通过小括号将这些内容括起作为一个对象，使用 ComputerName 属性并将结果保存在$computer 变量中。

如果在脚本执行时提供了参数-q，那么会出现$q 变量。这样可使用 funline 函数将计算机名作为标题输出，并使用 Powercfg 工具提供参数-query，从而得到本机当

前的详细电源计划。

如果提供了参数-c，则必须使用参数-t。这是因为$t 变量中包括的是时间信息，需要为-c 参数中指定的操作指定有限的超时时间，这样才不会无限期地等待。如果仅有$c 变量，则使用 throw 语句输出红色的错误信息，并停止脚本执行。

接下来脚本根据-c 参数值匹配 switch 语句中的分支，如果值为 mp，则使用包括在$t 变量值作为交流电情况下的显示器超时时间的分钟数；如果值为 mb，并且计算机在使用电池供电，则会使用$t 变量值设置显示器的超时时间；如果值为 db，则配置当前电源计划在达到$t 变量设定的分钟数后关闭驱动器；如果值为 sp，则在使用交流电的情况下当空闲时间超过$t 变量设定的分钟数后计算机将会进入待机状态；如果需要计算机休眠，则使用 hp，并修改用于交流电下休眠的电源计划值；如果使用 hb，则为电池模式下的休眠值。

此脚本在 Windows Server 2008 和 Windows XP 下通过-q 参数查询电源方案的结果如图 18-41 和图 18-42 所示。

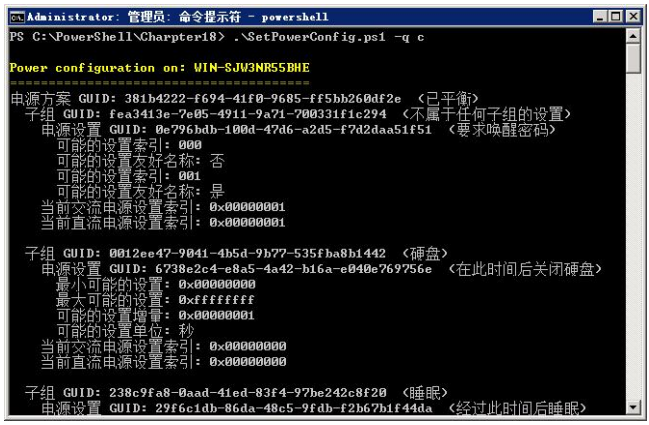


图 18-41　在 Windows 2008 中列出当前系统电源计划的详细方案

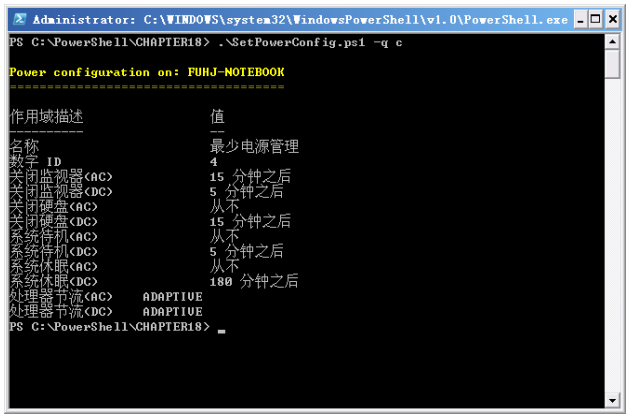


图 18-42　在 WindowsXP 中获取到的电源计划的信息

18.4 维护网络

18.4.1 使用网络设置

Windows Vista 开始在网络功能方面有了很大改善，包括新的防火墙服务及 IPv6 协议的增强支持等。同时从 Windows Vista 开始 WMI 中增加了很多用于操作防火墙和 IPv6 的新特性和计数器，可以显示和使用 IPv6 地址。

（1）查看网络设置

Windows Vista 和 Windows Server 2008 中包括强大的网络功能，允许用户以简单且便捷的方式操作网络。但是给网络管理员带来大量麻烦，如管理大量网络适配器，如图 18-43 所示。

图 18-43 需要管理大量网络适配器

为了有效地管理网络设备，例 18-17 创建名为“GetNetAdapterStatus.ps1”的脚本用于检测网络适配器的状态。

例 18-17 检测网络适配器的状态

```
param($computer="localhost",$help)
function funStatus($status)
{
 switch($status)
  {
   0 { " Disconnected" }
   1 { " Connecting" }
   2 { " Connected" }
   3 { " Disconnecting" }
   4 { " Hardware not present" }
   5 { " Hardware disabled" }
   6 { " Hardware malfunction" }
```

```
    7 { " Media disconnected" }
    8 { " Authenticating" }
    9 { " Authentication succeeded" }
    10 { " Authentication failed" }
   }
}
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: GetNetAdapterStatus.ps1
Produces a listing of network adapters and status on a local or remote machine.
PARAMETERS:
-computerName Specifies the name of the computer upon which to run the script
-help          prints help file
SYNTAX:
GetNetAdapterStatus.ps1 -computer WebServer
Lists all the network adapters and status on a computer named WebServer
GetNetAdapterStatus.ps1
Lists all the network adapters and status on local computer
GetNetAdapterStatus.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}
if($help) { "Printing help now..." ; funHelp }
$objWMI=Get-WmiObject -Class win32_networkadapter -computer $computer
funline("Network adapters and status on $computer")
foreach($net in $objWMI)
{
 Write-Host "$($net.name)"
  funstatus($net.netconnectionstatus)
}
```

为了获取网络适配器的状态，在该脚本中使用 Win32_NetWorkAdapter WMI 类返回状态代码。并创建了一个名为“funStatus”的函数，通过 switch 语句的代码块包括 Win32_NetWorkAdapterWMI 类定义的所有可能的状态代码。状态代码及其含义在 Windows 软件开发包（SDK）中有详细介绍，说明如下。

- 0：Disconnected（断开）。
- 1：Connecting（连接中）。
- 2：Connected（已连接）。
- 3：Disconnecting（断开中）。

- 4：Hardware not present（硬件不存在）。
- 5：Hardware disabled（硬件已禁用）。
- 6：Hardware malfunction（硬件故障）。
- 7：Media disconnected（媒介断开）。
- 8：Authenticating（权限认证中）。
- 9：Authentication succeeded（权限认证成功）。
- 10：Authentication failed（权限认证失败）。

在该脚本中通过 funStatus 函数将状态值转换为便于理解的内容，其执行结果如图 18-44 所示。

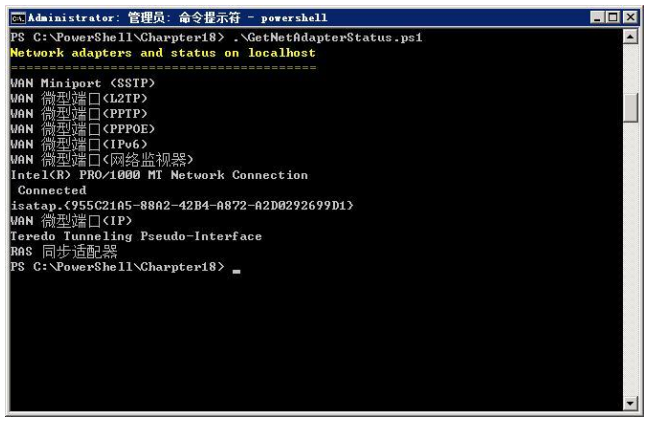


图 18-44　执行结果

（2）处理适配器配置

在处理所有适配器的状态后，还可以查询每个网络适配器的详细配置信息。可以通过选择“控制面板”|“网络和共享中心”|“网络连接”选项，打开“网络连接”窗口。在其中显示每个适配器的详细信息并做相应调整，如图 18-45 所示。

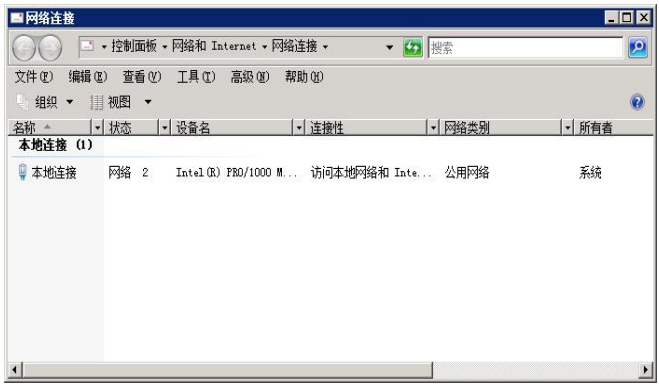


图 18-45　在“网络连接”窗口中查看每个适配器的详细信息

创建名为“GetNetAdapterConfig.ps1”的脚本收集特定网络适配器的用于排错

的详细信息，并且通过指定关键字仅返回有关网络适配器的特定配置信息，其代码如例 18-18 所示。

例 18-18　收集特定网络适配器的用于排错的详细信息

```
param($computer="localhost",$query,$help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: GetNetAdapterConfig.ps1
Produces a listing of network adapter configuration information
on a local or remote machine.
PARAMETERS:
-computer Specifies the name of the computer to run the script
-help       prints help file
-query      the type of query < ip, dns, dhcp, all >
SYNTAX:
GetNetAdapterConfig.ps1 -computerName WebServer
Lists default network adapter configuration on a
computer named WebServer
GetNetAdapterConfig.ps1 -computerName WebServer -query IP
Lists IPaddress, IPsubnet, DefaultIPgateway, MACAddress
on a computer named WebServer
GetNetAdapterConfig.ps1 -computerName WebServer -query DNS
Lists DNSDomain, DNSDomainSuffixSearchOrder, DNSServerSearchOrder,
DomainDNSRegistrationEnabled on a computer named WebServer
GetNetAdapterConfig.ps1 -computerName WebServer -query DHCP
Lists Index,DHCPEnabled, DHCPLeaseExpires, DHCPLeaseObtained,
DHCPServer on a computer named WebServer
GetNetAdapterConfig.ps1 -computerName WebServer -query ALL
Lists all network adapter configuration information on a computer
named WebServer
GetNetAdapterConfig.ps1 -help ?
Prints the help topic for the script
"@
$helpText
exit
}
if($help) { "Printing help now..." ; funHelp }
$class="win32_networkadapterconfiguration"
$IPproperty="IPaddress, IPsubnet, DefaultIPgateway, MACAddress"
$dnsProperty="DNSDomain, DNSDomainSuffixSearchOrder, `
 DNSServerSearchOrder, DomainDNSRegistrationEnabled"
$dhcpProperty="Index,DHCPEnabled, DHCPLeaseExpires, `
 DHCPLeaseObtained, DHCPServer"
if($query)
{
 switch($query)
 {
  "ip"     { $query="Select $IPproperty from $class" }
  "dns"    { $query="Select $dnsProperty from $class" }
  "dhcp"   { $query="Select $dhcpProperty from $class" }
  "all"    {
```

```
            $query = "Select * from $class" ; `
          Get-WmiObject -Query $query | format-list * ;
          exit
        }
    DEFAULT {
            $query = "Select * from $class" ; `
         Get-WmiObject -Query $query ; exit
        }
  }
}
ELSE
 {
  $query = "Select * from $class" ; `
  Get-WmiObject -Query $query ; exit
 }
Get-WmiObject -query $query | format-table [a-z]* -AutoSize
```

该脚本使用 param 语句定义了 3 个参数，即-computer、-query 及-help。其中设置-computer 的默认值为 localhost。如果用户未指定计算机，则默认返回本地计算机的网络配置。

接下来针对帮助信息创建的 funhelp 函数用于输出帮助信息，当用户未输入参数或输入错误的参数时，调用该函数提示输入的错误及正确的输入。脚本中使用 if 语句检查$help 变量是否存在，因为只有用户在调用脚本时指定了-help 参数，该变量才会存在。随后初始化用于 WMI 查询的变量，这些变量的存在是为了便于在后面对命令行应用中实现灵活调用。在用户输入的参数中包括-query 参数时，脚本将会调用 switch 语句判断$query 变量值。并执行相应的操作，输出 ip、dns、dhcp 和所有条目的信息。在$query 变量无输入的情况下还特别使用 ELSE 语句将 WMI 对象中的所有属性发送给 Get-WmiObject cmdlet 查询，在控制台输出当前计算机中的所有网络连接的信息。在代码的最后将 Get-WmiObject cmdlet 查询的结果通过管道传递给 Format-Table cmdlet 格式化输出，在使用 Format-Tables 时需要指定符合 Get-WmiObject cmdlet 输出代码格式的参数。该脚本的执行结果如图 18-46 和图 18-47 所示。

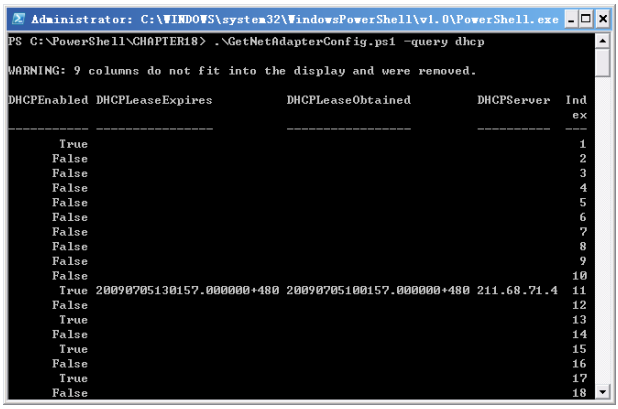


图 18-46　查询当前主机 DHCP 信息

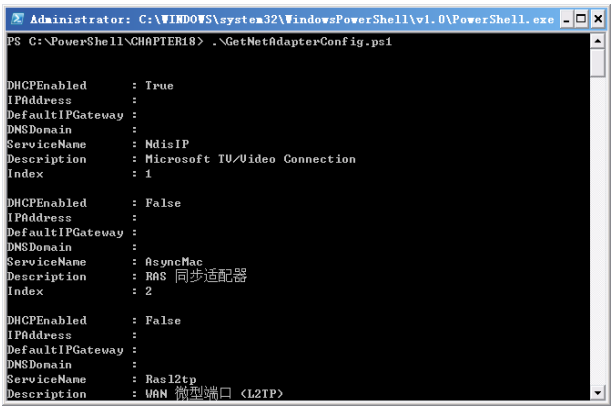


图 18-47　未指定检索项默认会输出所有的网络适配器信息

（3）筛选被赋值的属性

在前面脚本输出中可以看出在显示属性名称时未显示其值，从图 18-48 所示的 Windows 管理规范测试器（运行 WBEMTest.exe）中可以看到很多属性没有值。但是为了看到图中的 Win32_NetworkAdapterConfiguration 的对象编辑器需要在打开测试器之后连接的命名空间指向 root\CIMV2，在打开类别中输入 Win32_NetworkAdapterConfiguration 即可打开。

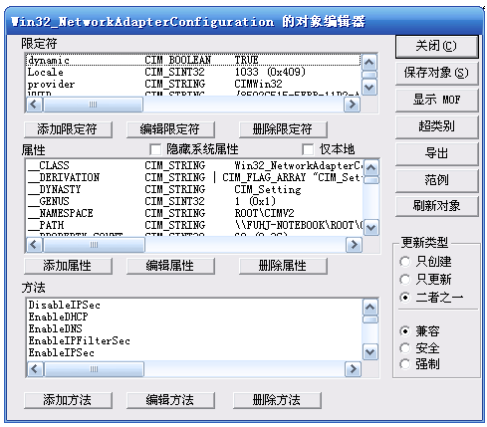


图 18-48　通过 Windows 管理规范测试器的对象编辑器可显示每个属性及值

为了有效地规范和整理输出的信息，例 18-19 创建名为“NetworkAdapterConfig Filtered.ps1”的脚本过滤无用属性。

例 18-19　过滤无用属性

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
```

```
}
Get-WmiObject win32_networkadapterconfiguration |
  foreach-object `
    {
     funLine("Querying: $($_.caption)")
      $_.psobject.properties |
   foreach-object `
        {
         If($_.value)
       {
         if ($_.name -match "__"){}
          ELSE
        {
          Write-Host "$($_.name)`t`t $($_.value)"
         }
   }
   }
}
```

该脚本通过 Get-WmiObject cmdlet 获取 win32_networkadapterconfiguration WMI 类的信息，随后使用管道将结果 Management 对象发送给 ForEach-Object cmdlet。这样即可遍历所有对象的属性，每次处理一个网络适配器。在 ForEach-Object cmdlet 代码块中需要为 WMI 信息设置标题，这样易于区分计算机中的不同网络适配器。在该脚本中通过判断 Value 属性是否存在确定是否输出该属性。为了程序的易读性，该脚本使用重音符来连接比较长的多行脚本内容，执行结果如图 18-49 所示。

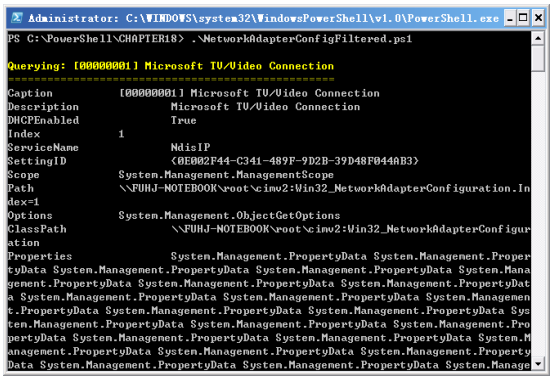


图 18-49　执行结果

18.4.2　设置网络适配器

如果计算机中有多个网络适配器，在 Windows 原有的 cmd 下配置其属性很困难，用户必须确保配置了适当的适配器并确认要禁用的不是正在使用中的网络适配器。本节介绍在处理多个网络适配器时可能出现的问题及其解决方法。

（1）检测多个网络适配器

对于 Windows 系统来说，操作系统会将无线网络放在所有网络连接中优先级最

高的网络使用。这对于普通用户可能是很方便的，一旦周围有无线网络就可以自动连接到无线网络中，便捷而有效。但是对于网络管理员来说这个特性可能会带来麻烦，甚至是安全问题。如出差在外的人员无法通过所在地的有线网络访问 Internet。Windows Vista 通常会建议启用无线网络适配器来上网，而如果听从了这个建议，并连接到不安全的网络，如图 18-50 所示，计算机可能存在很大的安全威胁。

图 18-50　连接到不安全的网络

创建一个名为“GetNetID.ps1”的脚本显示连接到本地计算机的网络适配器名称、接口索引编号、适配器信息及其介质形式，这些属性对于创建资产清单非常有用。该脚本的代码如下：

```
Get-WmiObject -Class win32_networkadapter |
format-table -Property name, interfaceIndex, `
adapterType, macAddress –autosize
```

该脚本中通过使用 Get-WmiObject cmdlet 并检索 Win32_NetWorkAdapter WMI 类信息，最后将输出的信息通过管道传递给 Format-Table cmdlet 格式化输出的内容，而仅限定输出 Name、InterfaceIndex、AdapterType 及 MacAddress 属性。-autosize 参数可以配合 Format-Table 使用，以进一步整理输出内容并调整显示方式，执行结果如图 18-51 所示。

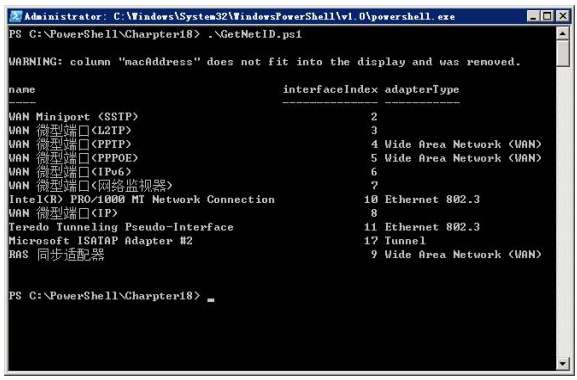


图 18-51　执行结果

（2）将网络适配器信息写入 Excel 文件

当系统中有多个网络适配器时通过控制台查看这些网络适配器的信息会显得信息量太大，无法细致地查看和处理，这时需要将信息写入到 Excel 文件中。例 18-20 创建名为“WriteNetworkAdapterInfoToExcel.ps1”的脚本获得安装在本地计算机上所有网络适配器的配置信息，并写入本地的 Excel 文件中便于后期分析，其代码如下：

例 18-20　获取网络适配器的配置信息

```
$strPath="C:\PowerShell\CHAPTER18\netAdapter.xls"
$objExcel=New-Object -ComObject Excel.Application
$objExcel.Visible=-1
$WorkBook=$objExcel.Workbooks.Add()
$sheet=$workbook.worksheets.item(1)
$x=2
$Computer = $env:computerName
$objWMIService = Get-WmiObject -class win32_NetworkAdapter `
   -computer $Computer
for($b=1 ; $b -le 10 ; $b++)
  {$sheet.Cells.item(1,$b).font.bold=$true}
$sheet.Cells.item(1,1)=("Name of Adapter")
$sheet.Cells.item(1,2)=("Interface Index")
$sheet.Cells.item(1,3)=("Index")
$sheet.Cells.item(1,4)=("DeviceID")
$sheet.Cells.item(1,5)=("AdapterType")
$sheet.Cells.item(1,6)=("MacAddress")
$sheet.Cells.item(1,7)=("netconnectionid")
$sheet.Cells.item(1,8)=("NetConnectionStatus")
$sheet.Cells.item(1,9)=("NetworkAddresses")
$sheet.Cells.item(1,10)=("PermanentAddress")
ForEach ($objNet in $objWMIService)
{
    $sheet.Cells.item($x, 1)=($objNet.Name)
    $sheet.Cells.item($x, 2)=($objNet.InterfaceIndex)
$sheet.Cells.item($x, 3)=($objNet.index)
$sheet.Cells.item($x, 4)=($objNet.DeviceID)
    $sheet.Cells.item($x, 5)=($objNet.adapterType)
$sheet.Cells.item($x, 6)=($objNet.MacAddress)
$sheet.Cells.item($x,7)=($objNet.netconnectionid)
    $sheet.Cells.item($x,8)=($objNet.NetConnectionStatus)
    $sheet.Cells.item($x,9)=($objNet.NetworkAddresses)
    $sheet.Cells.item($x,10)=($objNet.PermanentAddress)
    If($objNet.AdapterType -notMatch 'ethernet')
    {
     $sheet.Cells.item($x,5).font.colorIndex=3 # 32 is blue 16 silver/gray 8 is Aqua, 4 is
green, 3 is red
     $sheet.Cells.item($x,5).font.bold=$true
    }
$x++
}
 $range = $sheet.usedRange
 $range.EntireColumn.AutoFit()
IF(Test-Path $strPath)
```

```
    {
     Remove-Item $strPath
     $objExcel.ActiveWorkbook.SaveAs($strPath)
    }
  ELSE
    {
     $objExcel.ActiveWorkbook.SaveAs($strPath)
    }
```

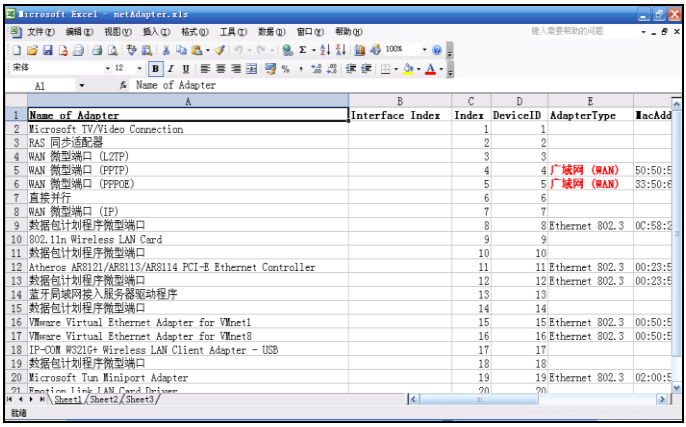
能够看到上述脚本中网络适配器信息的代码量很少，大多数代码主要是关于对 Excel 文件的操作方法。在通过 PowerShell 操作 Excel 之前需要指定 Excel 文件的存放路径，在该脚本中将其放在$strPath 变量中。随后创建一个 Excel.Application COM 对象的实例，用于创建和操作 Excel 表格。为操作 Excel 模块，将保存在$objExcel 变量中的 Excel.Application 对象的 Visible 属性设置为-1，表示窗口可见。然后在 Excel 中添加一个新的工作簿，打开第 1 个工作表并将引用信息保存在$sheet 变量中。为此需要引用新创建并保存在$workbook 变量中的新工作簿对象，然后使用 item 方法返回第 1 个工作表。

接下来声明变量$x，并赋值为 2，指定从第 2 行开始写。随后为了在 WMI 查询中能够使用计算机名，进入 env:\这个 PS 的驱动器中并获得用于保存计算机名的环境变量 computerName 的值保存在$Computer 变量中。其后使用 Get-WmiObject cmdlet 查询 Win32_NetworkAdapter WMI 类，查询得到的 Management 对象保存在$objwmiService 变量中。

下一段代码为从 WMI 中获得的每个属性提供标题行，这里使用 for 循环指定需要用粗体显示的行。为了将该行标题显示为粗体，将该字体的 Bold 属性设置为 True。设置标题后使用 Foreach 语句在 WMI 对象之间遍历，以找出所需的特定信息并将其插入到相应的列中。这里使用 item 方法来引用相应的单元格，需要为该方法提供横轴和纵轴的坐标信息，以使光标在单元格中定位。为了让操作过程简单而便捷，这里使用变量$x 跟踪要写入的行。通过改变其值来移动光标，特定的纵轴指定保存特定数据的列。检查网络适配器为以太网适配器或其他类型的最简单方法是检查 Win32_NetworkAdapter WMI 类中的 AdapterType 属性，该脚本的目的是标记非以太网适配器，所以使用-notmatch 操作符比较适配器类型。如果适配器类型不是“ethernet”（以太网），则改变字体的颜色并用粗体显示。处理每一条网络适配器记录后使用递增运算符++使$x 变量值递增，这样随后的一条适配器信息才可写入表格的下一行。

创建并显示整个表后，通常需要必要改变列宽以显示完整的信息。为此在该脚本中使用 AutoFit()方法用于表格内特定范围内的列对象，定义范围的简单方法是使用工作表对象的 UseRange 属性。

规整表格内容和格式之后，务必保存当前文档。如果工作簿已经存在，则将其删除，然后将工作表保存在新的工作簿中；否则直接将工作表保存为新工作簿。该脚本执行后的工作簿如图 18-52 所示。



	A	B	C	D	E	
1	Name of Adapter	Interface Index	Index	DeviceID	AdapterType	MacAdd
2	Microsoft TV/Video Connection		1	1		
3	RAS 同步适配器		2	2		
4	WAN 微型端口 (L2TP)		3	3		
5	WAN 微型端口 (PPTP)		4	4	广域网 (WAN)	50:50:5
6	WAN 微型端口 (PPPOE)		5	5	广域网 (WAN)	33:50:6
7	直接并行		6	6		
8	WAN 微型端口 (IP)		7	7		
9	数据包计划程序微型端口		8	8	Ethernet 802.3	0C:58:2
10	802.11n Wireless LAN Card		9	9		
11	数据包计划程序微型端口		10	10		
12	Atheros AR8121/AR8113/AR8114 PCI-E Ethernet Controller		11	11	Ethernet 802.3	00:23:5
13	数据包计划程序微型端口		12	12	Ethernet 802.3	00:23:5
14	蓝牙局域网接入服务器驱动程序		13	13		
15	数据包计划程序微型端口		14	14		
16	VMware Virtual Ethernet Adapter for VMnet1		15	15	Ethernet 802.3	00:50:5
17	VMware Virtual Ethernet Adapter for VMnet8		16	16	Ethernet 802.3	00:50:5
18	IP-COM W321G+ Wireless LAN Client Adapter - USB		17	17		
19	数据包计划程序微型端口		18	18		
20	Microsoft Tun Miniport Adapter		19	19	Ethernet 802.3	02:00:5

图 18-52　脚本执行后的工作簿

（3）识别已连接的网络适配器

多重网络环境在提供了便利的同时，也给计算机安全带来极大的威胁。计算机系统中多个网络一旦被桥接，而网络中又存在安全和不安全的网络，则恶意用户可以通过这种网桥攻击安全网络。

对于复杂网络环境下的计算机，多重网络环境在提供便利的同时，也给计算机带来了极大的威胁。计算机系统中多个网络一旦被桥接起来，而网络中又存在安全和不安去的网络，恶意用户可以通过这种网桥对安全网络进行攻击。这就会给安全网络造成很大的威胁。

为了避免由于复杂网络环境下可能由网络适配器桥接造成的安全风险，创建名为“FindConfigurationOfConnectedAdapters.ps1”的脚本识别连通多个计算机的网络适配器。该脚本仅返回已经连通的网络适配器的数据，如果没有活动连接，则不会返回任何数据。该脚本的代码如下：

```
$computer="localhost"
$connected=2
Get-WmiObject -Class win32_networkadapter -computername $computer `
-filter "netconnectionstatus = $connected" |
foreach-object `
 {
   Get-WmiObject -Class win32_networkadapterconfiguration `
   -computername $computer -filter "Index = $($_.deviceID)"
 }
```

该脚本使用了两个 WMI 类，其中 Win32_NetworkAdapter WMI 类具有名为“Connected”的属性，而 Win32_NetworkAdapterConfiguration WMI 类则无。

该脚本首先定义变量$compute 用于执行 WMI 查询，$connected 用于显示 NetConnectionStatus 属性值表示计算机是否已经连接。随后查询 Win32_NetworkAdapter 类，这样可获得一个可以代表网络适配器是否连接的

Management 对象。如果只希望已经连接的网络适配器的信息，则可以使用-filter 参数。这些信息会通过管道命令发送到 Foreach-Object cmdlet，在其中查询 Win32_NetworkAdapterConfiguration 类。并使用筛选器查询上文中筛选后的网络适配器，适配器可以通过当前管道对象的 DeviceID 识别。

（4）设置静态 IP 地址

很多时候，网络管理员需要为网络设备指定静态 IP，以便有效地管理服务器。尽管在如图 18-53 所示的“Internet 协议（TCP/IP）属性”对话框中很容易设置静态 IP 地址，但是设置大量服务器则比较麻烦。

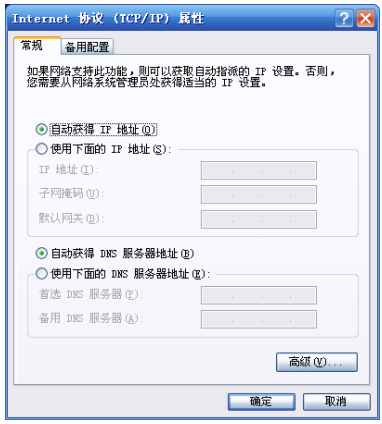


图 18-53　“Internet 协议（TCP/IP）属性”对话框

PowerShell 可以使用 Win32_NetworkAdapterConfiguration WMI 类中包括的 14 个方法，创建名为“SetStaticIP.ps1”的脚本中演示其中的 3 个方法。

```
param($computer="localhost",$q,$ip,$sm,$dg,$dns,$help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: SetStaticIP.ps1
Sets a static IP address on a local or remote machine.
PARAMETERS:
-computerName Specifies the name of the computer upon which to run the script
-q            Queries all IP bound network adapters
-ip           IP address to use
-sm            Subnet mask to use
-dg           Default gateway to use
-dns          Dns server to use
-help         prints help file
SYNTAX:
SetStaticIP.ps1 -q "yes" -computer WebServer
Lists all the network adapters bound to IP on a computer named WebServer
SetStaticIP.ps1
Lists all the network adapters bound to IP on local computer
SetStaticIP.ps1 -ip "10.0.0.1" -sm "255.0.0.0" -dg "10.0.0.5" -dns "10.0.0.2"
Sets the Ip address to 10.0.0.1 and the subnet mask to 255.0.0.0 and the default
```

```
Gateway to 10.0.0.5 with a dns server of 10.0.0.2 on the local machine
SetStaticIP.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
function FunEvalRTN($rtn)
{
Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "No errors for $strCall" }
   66 { Write-Host -foregroundcolor red "$strCall reports" `
        " invalid subnetMask" }
   70 { Write-Host -ForegroundColor red "$strCall reports" `
        " invalid IP" }
   71 { Write-Host -ForegroundColor red "$strCall reports" `
         " invalid gateway" }
   91 { Write-Host -ForegroundColor red "$strCall reports" `
         " access denied"}
   96 { Write-Host -ForegroundColor red "$strCall reports" `
         " unable to contact dns server"}
   DEFAULT { Write-Host -ForegroundColor red "$strCall service reports" `
             " ERROR $($rtn.returnValue)" }
  }
  $rtn=$strCall=$null
}

if($help) { funhelp }
if($q)
{
 Get-WmiObject -Class win32_networkadapterconfiguration `
 -computer $computer -filter "ipenabled = 'true'"
exit
}
if(!$ip)   { funhelp }
if(!$sm)   { funhelp }
if(!$dg)   { funhelp }
if(!$dns) { funhelp }
$global:RTN = $null
$metric = [int32[]]1
$objWMI = Get-WmiObject -Class win32_networkadapterconfiguration `
 -computer $computer -filter "ipenabled = 'true'"
$RTN=$objwmi.EnableStatic($ip, $sm)
$strCall="enable static IP and subnet mask"
FunEvalRTN($rtn)
$RTN=$objwmi.SetGateways($dg, $metric)
$strCall="enable set default gateway and metric"
FunEvalRTN($rtn)
$RTN=$objwmi.SetDNSServerSearchOrder($dns)
$strCall="Set the dns server search order"
FunEvalRTN($rtn)
```

在脚本中定义了 FunEvalRTN 函数，用于判断调用不同的 WMI 方法以配置 IP 地址后返回的代码。在该函数内部使用 switch 语句判断返回代码的 ReturnValue 属

性，如果 ReturnValue 的值是 0，表示没有出错。而其他值均代表命令没有成功完成。

- 0：没有错误，正常完成操作。
- 66：错误的子网掩码。
- 70：非法 IP。
- 71：非法的网关。
- 91：访问被拒绝。
- 96：不能连接到 DNS 服务器。
- default 表示出现错误，但错误不在上述情况。

如果希望返回的字符串包括更多信息，那么可以包括一个变量$strCall，其中包括生成 ReturnValue 调用的方法名称。

该脚本会检查-q 参数，如果存在该参数堆栈中会存在$q 变量，脚本运行 WMI 查询并显示所有启用 IP 地址的网络适配器信息。在设置相关网络参数时要求相关参数不能为空；否则将无法设置 IP 信息，并调用 funhelp 函数获得帮助信息。

随后使用$global 标记生成全局变量，并使其名称成为全局性的。为了以数组形式保存存在的网关地址，在脚本中使用[int32]类型约束符，以保证输入的数字是 int32 数据类型。然后在类型约束符内部插入一对空的方括号标示对约束符的调用，接下来将数组指定给方法调用中使用的变量$metric。此脚本的执行结果如图 18-54 所示。

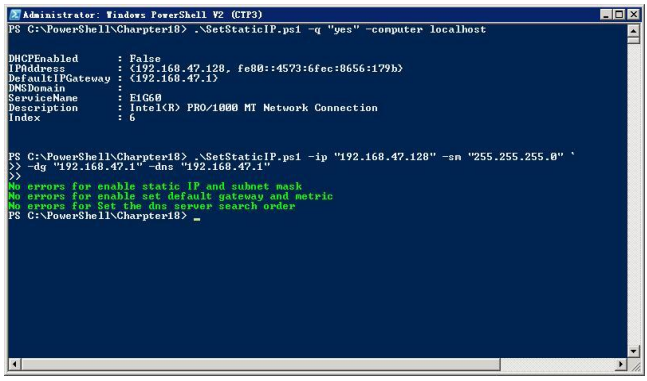


图 18-54　执行结果

由于 Windows Vista 和 Windows Server2008 与 Windows XP 之间的网络 WMI 模型已经存在很大的区别，并且该脚本中使用的 EnableStatic()、SetGateways()和 SetDNSServerSearchOrder()等方法均是 Vista 以后操作系统中的新增方法，所以这个脚本仅适用于 Vista 以上版本的 Windows 操作系统中的 PowerShell。

（5）启用 DHCP

除了可以对网络适配器绑定 IP 外，还可以使用 DHCP（Dynamic Host Configuration Protocol）服务器获取 IP 地址，启用 DHCP 的目的就是为了防止由于静态地址绑定而造成对网络地址的大量占用。DHCP 的工作原理一般分为 4 步，即

客户端查找 DHCP Server、Server 提供 IP 租用地址、客户端接受 IP 租约和租约确认。服务器之类的计算机需要指定 IP 地址；否则一旦 IP 地址被更换，用户将无法连接到服务器。比较好的策略就是为服务器预留一定得地址段，为服务器绑定 IP 地址，而 PC 则可以使用 DHCP 获取 IP 地址。

例 18-21 创建名为“WorkWithDHCP.ps1”的脚本用于报告 DHCP 状态、启用 DHCP、释放由 DHCP 分配的 IP 地址，并更新由 DHCP 分配的 IP 地址。

例 18-21　控制 DHCP

```
param($computer="localhost",$action,$help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: WorkWithDHCP.ps1
Works with DHCP settings on a local or remote machine.
PARAMETERS:
-computerName Specifies the name of the computer upon which to run the script
-action <q(uery) e(nable) r(elease) rr(release/renew)   action to perform
-help    prints help file
SYNTAX:
WorkWithDHCP.ps1 -q "yes" -computer WebServer
Queries DHCP settings on a computer named WebServer
WorkWithDHCP.ps1 -action e
enables DHCP on local computer
WorkWithDHCP.ps1 -action r
Releases the DHCP address on the local machine
WorkWithDHCP.ps1 -action rr
Releases and then renews the DHCP address on the local machine
WorkWithDHCP.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
function FunEvalRTN($rtn)
{
Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "No errors for $strCall" }
   82 { Write-Host -foregroundcolor red "$strCall reports" `
        " Unable to renew DHCP lease" }
   83 { Write-Host -ForegroundColor red "$strCall reports" `
        " Unable to release DHCP lease" }
   91 { Write-Host -ForegroundColor red "$strCall reports" `
         " access denied"}
   DEFAULT { Write-Host -ForegroundColor red "$strCall service reports" `
             " ERROR $($rtn.returnValue)" }
  }
  $rtn=$strCall=$null
}
if($help) { funhelp }
```

```
$global:RTN = $null
if(!$action) { $action="q" }
$objWMI = Get-WmiObject -Class win32_networkadapterconfiguration `
 -computer $computer -filter "ipenabled = 'true'"
Switch($action)
{
 "e" {
       $rtn = $objWMI.EnableDHCP() ;
  $strCall = "Enable DHCP" ;
  FunEvalRTN($rtn)
     }
  "r" {
       $rtn = $objWMI.ReleaseDHCPLease() ;
  $strCall = "Release DHCP address" ;
  FunEvalRTN($rtn)
     }
 "rr" {
       $rtn = $objWMI.RenewDHCPLease() ;
  $strCall = "Release and Renew DHCP address" ;
  FunEvalRTN($rtn)
     }
 "q" {
       "DHCP Server: $($objWMI.dhcpserver)"
   "Lease obtained: " + [Management.ManagementDatetimeConverter]::`
    todatetime($objWMI.DHCPleaseObtained)
   "Lease expires: " + [Management.ManagementDatetimeConverter]::`
   todatetime($objWMI.DHCPleaseExpires)
 }
}
```

该脚本首先使用 param 语句定义 3 个参数，其中-computer 参数的默认值为 localhost。然后声明全局变量 RTN，并将其设置为$null，用于在代码段之间传递参数。如果没有提供任何参数，则显示当前 DHCP 的配置信息。如果$action 变量没有参数，则默认为 q 参数，用于检索现有 DHCP 信息。

接下来使用 switch 语句对判断$action 变量值，如果为 e，则在目标计算机上启用 DHCP。这里调用 enableDHCP()方法为$strCall 变量指定一个字符串，并将其传递给 FunEvalRTN 函数判断 enableDHCP()方法是否成功执行；如果为 r，则调用 releaseDHCP()方法释放 DHCP 地址，并返回值$strCall 传递给 FunEvalRTN 函数判断是否执行成功；如果为 rr，则更新 DHCP 地址为$strCall 变量指定字符串，然后判断返回值。

在 switch 语句的最后需要显示提供当前 IP 地址的 DHCP 服务器的地址，并查询获得租约的时间和过期时间。需要将 UTC 事件对象转换为客户端所需的正常时间，为此可以使用 Manangement.ManangementDateTimeConverter 这个.NET 框架的类，并调用其中的 toDateTime 静态方法将 UTC 格式的日期时间对象传递给该方法。该脚本的执行结果如图 18-55 所示。

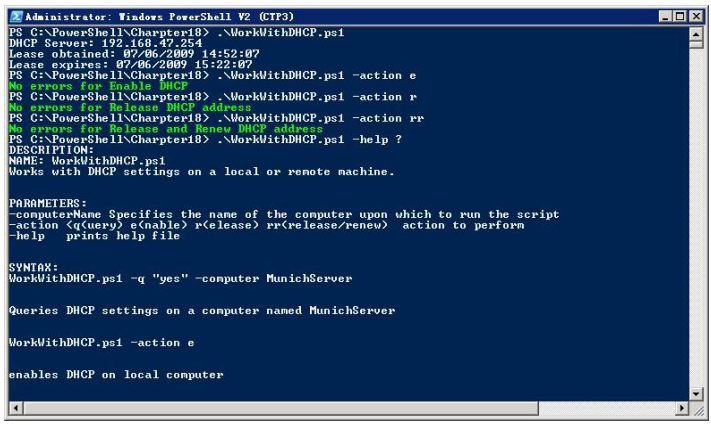


图 18-55　查询和更新 DHCP 信息

上面脚本中使用的 EnableDHCP()、ReleaseDHCPLease()和 RenewDHCPLease()等方法均是 Vista 以后的操作系统新增的方法，该脚本仅适用于 Windows Vista 及其以上版本。

18.4.3　配置 Windows 防火墙

Windows Vista 和 Windows Server 2008 在安全方面的最大改进是使用了全新的 Windows 防火墙系统。如图 18-56 所示。

图 18-56　Windows Vista 和 Windows Server 2008 的增强防火墙系统

可以通过 netsh 命令管理防火墙。

（1）查看防火墙设置

对于网络安全来说，允许或禁止哪些程序向外通信是很重要的信息，没有正确的配置的防火墙对于用户是个累赘。很多软件包括未知或不必要的服务，可能会在安装过程中在防火墙上打开某个端口。例 18-22 创建名为“ParseFWConfig.ps1”脚本可以检测到这些端口。

例 18-22　检测端口

```
$fwCfg = netsh firewall show config

$enable=$disable=$null
switch -regex ($fwCfg)
 {
  "启用" #此处的内容会根据操作系统语言的不同而不同，中文 Win"启用"，英文"enable"
    {
 $enable+=$switch.current+"`n"
}
  "禁用" #此处的内容会根据操作系统语言的不同而不同，中文 Win"禁用"，英文"disable"
    {
 $disable+=$switch.current+"`n"
}
 }
Write-Host -ForegroundColor cyan `
  "Firewall configuration on $env:computername"
Write-Host -ForegroundColor green `
  "The following are enabled`n"
  $enable
Write-Host -ForegroundColor red `
  "The following are disabled`n"
  $disable
```

此脚本借助 netsh 工具显示 Windows 防火墙的配置信息，并将结果信息保存在$fwCfg 变量中。初始化$enable 和$disable 变量为$null，然后使用 switch 语句执行正则表达式来匹配保存在$fwCfg 变量中的对象。搜索满足“启用/禁用”关键字的内容并将匹配结果的当前行添加到$enable/$disable 变量中，并且在行末添加换行符。最后检索 env:\这个 ps 驱动器中的计算机名，将其作为报告的标题输出，执行结果如图 18-57 所示。

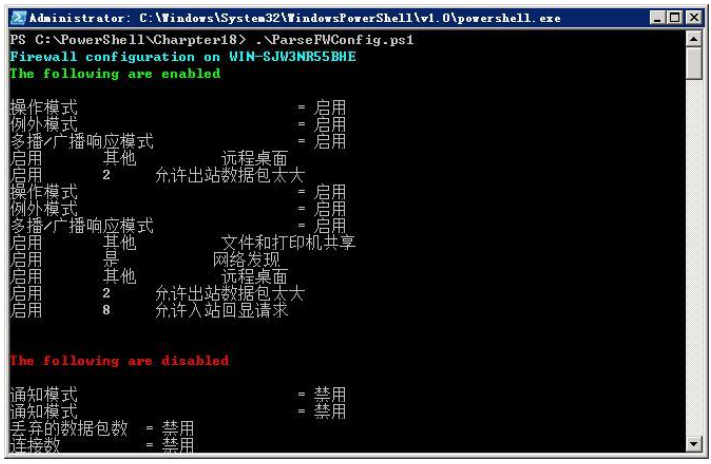


图 18-57　执行结果

（2）设置防火墙选项

Windows Vista 和 Windows Server 2008 系统，一般需要设置防火墙的远程管理

和共享文件夹选项。

例 18-23 创建名为“EnableRemoteAdmin.ps1”的脚本用于启用远程管理。

例 18-23　启用远程管理

```
$errRTN=netsh firewall set service remoteAdmin enable
if($errRTN -match '确定')    #此处根据返回值判定，中文 win 搜索'确定'，英文 win 搜索'ok'
   { Write-Host -ForegroundColor green "Remote admin enabled" }
   ELSEIF($errRTN -match 'requires elevation')
   { Write-Host -ForegroundColor red "Remote admin not enabled" `
     "The operation requries admin rights"}
ELSE
 { Write-Host -ForegroundColor red "Remote admin not enabled" `
   "The error reported was $errRTN" }
```

其中使用 netsh 工具及 set service 命令启用远程管理服务，并将结果信息传递给变量，通过从变量中查找“确定”（在英文操作系统中搜索“ok”）字符串来确定是否启动服务成功。由于 Windows 防火墙允许响应的端口必须具有管理员权限；否则在输出内容中查找“requires elevation”字符串，并提示用户需要提升权限来执行此命令。除了上除两种可能性以外，该脚本将错误原因保存在$errRTN 变量中输出。

启动共享文件服务使用 netsh 启用 fileAndPrint 服务，之后将错误信息存放在$errRTN 变量中输出。例 18-24 创建名为“EnableShareFolders.ps1”的脚本实现该功能。

例 18-24　输出错误原因

```
$errRTN=netsh firewall set service fileAndPrint enable
if($errRTN -match '确定')#此处根据返回值判定，中文 win'确定'，英文 win'ok'
   { Write-Host -ForegroundColor green "Shared folders enabled" }
   ELSEIF($errRTN -match 'requires elevation')
   { Write-Host -ForegroundColor red "Shared folders not enabled" `
     "The operation requries admin rights"}
ELSE
 { Write-Host -ForegroundColor red "Shared folders not enabled" `
   "The error reported was $errRTN" }
```

18.5　Windows 排错

18.5.1　启动故障排错

如果 Windows Vista 和 Windows Server 2008 无法正常启动，则可以检查引导配置文件是否出现错误；另外可以检查启动服务及其依存性。Windows 中的一些服务依赖于其他服务、系统驱动程序和组件的加载顺序。如果系统组件被停止或运行不正常，则依赖于它的服务会受到影响。

（1）检查引导配置文件

检查运行 Windows Vista 和 Windows Server 2008 的计算机引导配置文件通常可以为用户解决引导有关的问题，提供很多有价值的信息。类似引导分区、引导目录，以及 Windows 目录等信息往往都对排除故障很有用，大多数情况下通过 VBS 脚本获取这些信息需要花费很多时间。

例 18-25　读取引导配置

```
param($computer="localhost", [switch]$help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: DisplayBootConfig.ps1
Displays a boot up configuration of a Windows system
PARAMETERS:
-computer     The name of the computer
-help         prints help file
SYNTAX:
DisplayBootConfig.ps1 -computer WebServer
Displays boot up configuration of a computer named WebServer
DisplayBootConfig.ps1
Displays boot up configuration on local computer
DisplayBootConfig.ps1 -help
Displays the help topic for the script
"@
$helpText
exit
}
if($help){ "Obtaining help ..." ; funhelp }
$wmi = Get-WmiObject -Class win32_BootConfiguration `
      -computername $computer
format-list -InputObject $wmi [a-z]*
```

该脚本使用 param 语句定义了$computer 和$help 变量，前者的默认值为 localhost。设置-help 参数为 switch，即在使用该参数时不需要提供额外信息，并且使用 Get-WmiObject cmdlet 从 Win32_BootConfiguration WMI 类中获取信息。如果需要，可以将$computer 变量中的值提供给-computername 参数。这样可以通过 Get-WmiObject cmdlet 连接到远程计算机，最终将返回的 management 对象传递给 Format-List cmdlet。使用范围运算符（range operator）[a-z]*选择字符开头的属性，以过滤报告中的所有系统属性（因为系统属性均以下画线开头）。

该脚本的执行结果如图 18-58 所示。

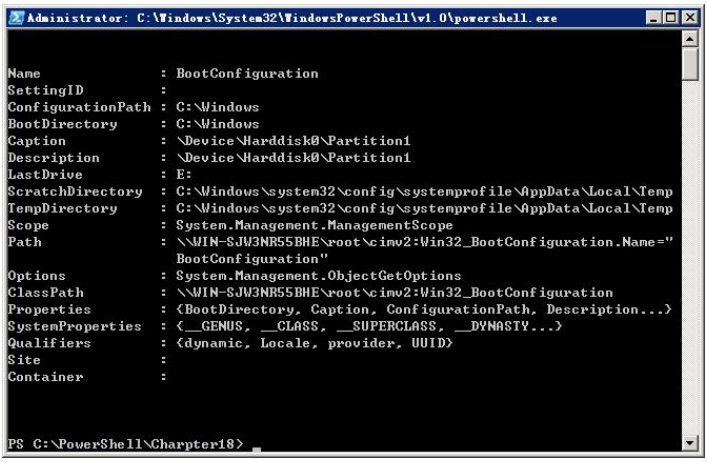


图 18-58　获取 Windows 引导配置信息

（2）检查启动进程

在 Windows Vista 和 Windows Server 2008 系统中有部分程序伴随系统启动，它们以不同启动组的形式存在。很多恶意软件和病毒以这种形式启动，当操作系统启动出现问题时需要检查这些启动组。

创建名为“DetectStartupPrograms.ps1”的脚本显示本地或远程计算机的自动运行程序的状态，并查看基本或完整的程序信息，其代码如下：

```
param($computer="localhost", [switch]$full, [switch]$help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: DetectStartUpPrograms.ps1
Displays a listing of programs that automatically start
PARAMETERS:
-computer     the name of the computer
-full         prints detailed information
-help         prints help file
SYNTAX:
DetectStartUpPrograms.ps1 -computer WebServer -full
Displays name, command, location, and user information
about programs that automatically start on a computer named WebServer
DetectStartUpPrograms.ps1 -full
Displays name, command, location, and user information
about programs that automatically start on the local computer
DetectStartUpPrograms.ps1 -computer WebServer
Displays a listing of programs that automatically start on a computer named WebServer
DetectStartUpPrograms.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
```

```
if($help){ "Obtaining help ..." ; funhelp }
if($full)
 { $property = "name", "command", "location", "user" }
else
 { $property = "name" }
Get-WmiObject -Class win32_startupcommand -computername $computer |
Sort-Object -property name  |
format-list -property $property
```

该脚本中使用 param 语句定义了$computer、$full 和$help 变量，分别用于指定脚本作用的计算机及帮助信息。随后定义了两个 switch 参数，其中-full 用于输出完整的启动程序信息；-help 用于输出帮助信息。

如果在脚本运行时提供了-full 参数，则输出程序的名称、可执行文件、路径及用户名等信息；否则仅显示名称。脚本中通过调用 Get-WmiObject cmdlet 获得所有的自动运行程序，将结果对象用管道发送给 Sort-Object cmdlet，同时分类属性名称。最后使用 Format-List cmdlet 选择由$property 变量指定的属性输出，该脚本的执行结果如图 18-59 所示。

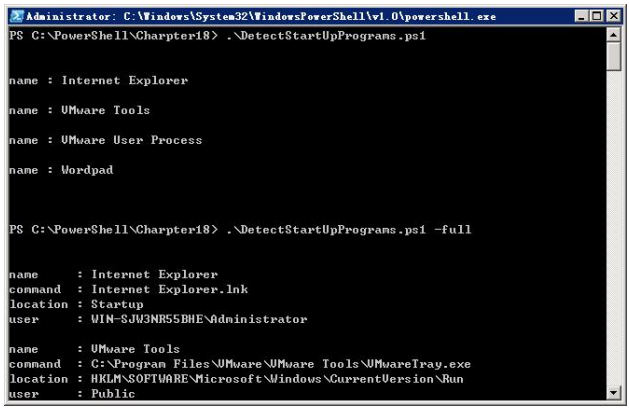


图 18-59　执行结果

18.5.2　查看服务依存性

Windows 中的系统服务具有依存性，如果一个服务未启动，可能导致更多依赖于它的服务均无法启动。以 Base Filtering Engine 服务为例，其基本筛选引擎（BFE）是一种管理防火墙和 Internet 协议安全（IPsec）策略及实施用户模式筛选的服务，停止或禁用 BFE 服务将大大降低系统的安全，并造成 IPsec 管理和防火墙应用程序产生不可预知的行为。在 Windows Server 2008 中，这个服务依赖于 Remote Procedure Call（RPC）服务，而 IKE and AuthIP IPsec Keying Modules、Internet Connection Sharing（ICS）、IPsec Policy Agent、Routing and Remote Access 和 Windows Firewall 这 5 个服务依赖于它。使用服务依存性有利于开发人员直接使用系统中的任何功能，降低开发工作量。与此同时带来的缺点是很难判断一些看似没有任何相互没有关联的服

务之间的关系，在图形界面中可以通过服务控制台了解这些信息。为此双击服务列表中的一个服务，然后打开“依存关系”选项卡，如图 18-60 所示。

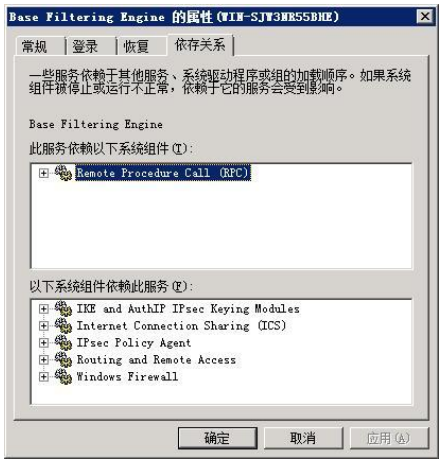


图 18-60　“依存关系”选项卡

为在非图形界面，如远程或者 Windows Server 2008 Server Core 下查看服务的依存关系，便于进一步查找和排除错误，例 18-26 创建名为“ServiceDependencies.ps1”的脚本。

例 18-26　查看服务的依存关系

```
    $erroractionpreference = "SilentlyContinue" # hides any cryptic error messages due to
security
    Param($computer = "localhost", [switch]$help)
    function funline ($strIN)
    {
     $num = $strIN.length
     for($i=1 ; $i -le $num ; $i++)
      { $funline = $funline + "=" }
        Write-Host -ForegroundColor yellow $strIN
        Write-Host -ForegroundColor darkYellow $funline
    }
    function funHelp()
    {
    $helpText=@"
    DESCRIPTION:
    NAME: ServiceDependencies.ps1
    Displays a listing of services and their dependencies
    PARAMETERS:
    -computer      The name of the computer
    -help          prints help file
    YNTAX:
    ServiceDependencies.ps1 -computer WebServer
    Displays a listing of services and their dependencies on a computer named WebServer
    ServiceDependencies.ps1
    Displays a listing of services and their dependencies on the local machine
    ServiceDependencies.ps1 -help ?
    Displays the help topic for the script
```

```
"@
$helpText
exit
}
if($help){ "Obtaining help ..." ; funhelp }
$dependentProperty = "name", "displayname", "pathname",
                          "state", "startmode", "processID"
$antecedentProperty = "name", "displayname",
                           "state", "processID"
if($computer = "localhost") { $computer = $env:computername }
funline("Service Dependencies on $($computer)")
New-Variable -Name c_padline -value 14 -option constant # allows for length of

displayname
Get-WmiObject -Class Win32_DependentService -computername $computer |
Foreach-object `
 {
  "=" * ((([wmi]$_.dependent).pathname).length + $c_padline)
  Write-Host -ForegroundColor blue "This service:"
    [wmi]$_.Dependent |
      format-list -Property $dependentProperty
  Write-Host -ForegroundColor cyan "Depends on this service:"
    [wmi]$_.Antecedent |
      format-list -Property $antecedentProperty
        "=" * ((([wmi]$_.dependent).pathname).length + $c_padline) + "`n"
 }
```

因为某些服务即使使用本地管理员组账户登录也无法访问，而且有些服务是系统自身运行时使用的服务，不允许用户操作，所以为了避免执行脚本时出现错误，该脚本首先将$erroractionpreference 自动变量赋值为 SilentlyContinue。

接下来使用 param 语句定义命令行-computer 参数指定运行这个脚本的主机和在需要时显示帮助信息-help 参数，并使用了之前脚本中定义过的 funline 函数输出标题信息。在后面定义了$dependentProperty 和$antecedentProperty 变量分别保存属性列表，在执行 WMI 类查询时可以得到与这两个变量对应的属性值。在 param 语句中的$computer 变量值是 localhost，默认指向本机。如果在调用此脚本时传递了-computer 参数，则默认使用该参数。如果用户没有提供计算机名，则需要将 localhost 翻译为本机的主机名，主机名可以通过查询 PS 驱动器的$env 变量获取。随后使用 New-Variable cmdlet 创建 c_padline 常量，并使用-option 参数，New-Variable cmdlet 的-name 参数不需要变量名以美元符（$）开头。

最后使用 Get-WmiObject cmdlet 查询 WMI 类 Win32_DependentService，这个 WMI 类是个关联类，与另外两个 WMI 类，即 Win32_BaseService 和 Win32_BaseService 有关（注意这不是错误，这个类可以关联自身，通过这种方式可以知道服务之间的依存关系）。可以使用 Get-WmiObject cmdlet 的-computername 参数使脚本具有查询本地或远程 WMI 的能力，命令结尾使用管道对象将处理后的结果对象传递给 ForEach-Object cmdlet。随后使用[WMI]management 对象获得有关服

务依存性的信息，并将结果 management 对象用管道发送给 Format-List cmdlet 输出保存在$dependentProperty 变量中的所有结果。

该脚本的执行结果如图 18-61 所示。

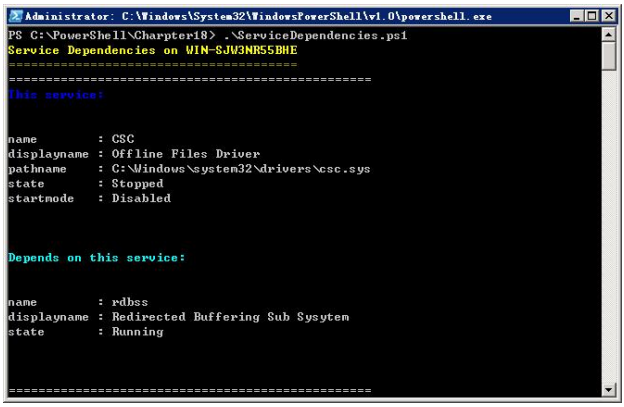


图 18-61　执行结果

（1）检查设备驱动

设备驱动和服务功能类似，可以自动运行并提供一定功能。只是设备驱动更接近于硬件底层，并不像服务那样容易发现和检查。设备驱动一旦出现问题，往往伴随某种设备功能的失灵，所以对于系统管理员来说检查设备驱动也很重要。

例 18-27　检查硬件驱动

```
    param($computer="localhost", $a="h", [switch]$help)
    function funHelp()
    {
    $helpText=@"
    DESCRIPTION:
    NAME: CheckDeviceDrivers.ps1
    Displays a listing of system drivers that are set to
    automatic, manual, boot, system or all drivers
    PARAMETERS:
    -computer     The name of the computer
    -a(ction)     < a(ll), r(unning), s(topped), b(oot),m(anual), au(to), sy(stem), h(elp) >
    -help         prints help file
    SYNTAX:
    CheckDeviceDrivers.ps1 -computer WebServer -a b
    Displays a listing of all device drivers that are set to start on boot on a computer named
WebServer
    CheckDeviceDrivers.ps1 -a auto
    Displays a listing of all device drivers on local computer set to start up automatically
    CheckDeviceDrivers.ps1 -computer WebServer -a m
    Displays a listing of all device drivers
    that are set to start manually on a computer named WebServer
    CheckDeviceDrivers.ps1 -help ?
    Displays the help topic for the script
    "@
    $helpText
```

```
exit
}
if($help){ "Obtaining help ..." ; funhelp }
switch($a)
{
 "a" {
       "Retrieving all device drivers"
  $filter = "started = 'true' or started = 'false'"
 }
 "r" {
       "Retrieving all running device drivers"
  $filter = "started = 'true'"
 }
 "s" {
       "Retrieving all stopped device drivers"
  $filter = "started = 'false'"
 }
 "b" {
       "Retrieving boot device drivers"
  $filter = "startmode = 'boot'"
 }
 "m" {
       "Retrieving manual device drivers"
  $filter = "startmode = 'manual'"
 }
 "au" {
       "Retrieving auto device drivers"
  $filter = "startmode = 'auto'"
 }
 "sy" {
       "Retrieving system device drivers"
  $filter = "startmode = 'system'"
 }
 "h" {
       "You need to specify an action. The -a parameter is required"
  "Try this: " +   $MyInvocation.MyCommand.Definition + " -h"
  exit
 }
 DEFAULT
       {
       "You need to specify an action. The -a parameter is required"
  "Try this: " +   $MyInvocation.MyCommand.Definition + " -h"
  exit
 }
}
$wmi = Get-WmiObject -Class win32_systemdriver `
      -computername $computer -filter $filter
format-table -InputObject $wmi -property `
       displayname, pathname, name   -autosize
```

该脚本通过 param 语句定义了-computer，-a 和-help 参数。其中-a 指定要执行的操作，默认为 h，即显示帮助信息。

该脚本通过 switch 语句定义了主要操作的业务逻辑，允许用户指定一系列不同且相关的设备驱动。switch 语句显示一条当前操作状态信息，然后通过设定$filter

变量值，为 Get-WmiObject cmdlet 创建 filter（筛选）参数。

该脚本中调用了可以在多个 switch 语句中使用的$MyInvocation.MyCommand.Definition 命令，用于在输出信息中引用正在运行的脚本。其核心功能是通过 Get-WmiObject cmdlet 查询 Win32_SystemDriver WMI 类，通过指定$computer 变量查询指定的计算机，然后可以使用通过 switch 语句创建的筛选器并将查询结果保存在$wmi 变量中。为了在输出时格式化该变量，使用了带-InputObject 参数的 Format-Table cmdlet。同时保存$wmi 变量中的 management 对象，执行结果如图 18-62 所示。

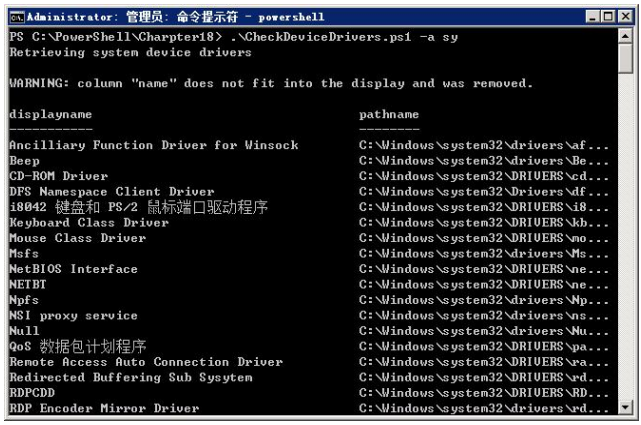


图 18-62　执行结果

（2）检查启动服务

在 Windows 中有些服务随系统启动，如果其中的某个服务无法启动，则可能导致系统不稳定或其他不可预知的结果。如果服务出错，首先需要检查服务。将其按照启动类型排列。然后查找所有停止自动运行的服务，如图 18-63 所示。

图 18-63　检查未启动的自动运行服务是排错的基本步骤

为了便于在脚本中查询未启动的自动运行服务，例 18-28 创建名为“AutoService

NotRunning.ps1”的脚本。

例 18-28　查询未启动的自动运行服务

```
param($computer="localhost", [switch]$help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: AutoServicesNotRunning.ps1
Displays a listing of services that are set to automatic, but are not presently running
PARAMETERS:
-computer     The name of the computer
-help         prints help file
SYNTAX:
AutoServicesNotRunning.ps1 -computer WebServer
Displays a listing of all non running servicesthat are set to automatically start on a
computer named WebServermunich
AutoServicesNotRunning.ps1
Displays a listing of all services that are set to automatic, but are not presently running on
the local machine
AutoServicesNotRunning.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
if($help){ "Obtaining help ..." ; funhelp }
$wmi = Get-WmiObject -Class win32_service -computername $computer `
   -filter "state <> 'running' and startmode = 'auto'"
if($wmi -eq $null)
  { "No automatic services are stopped" }
Else
  {
   "There are $($wmi.count) automatic services stopped.
         The list follows ... "
     foreach($service in $wmi) { $service.name }
   }
```

该脚本使用 Get-WmiObject cmdlet 查询 Win32_Service WMI 类，通过自定义仅返回设置为自动运行服务器的当前状态，输出信息说明其是否正常。通过指定-computername 参数选择本地或远程计算机，使用-filter 参数减少返回的 Win32_Service 类的实例数量。因为只需要知道启动类型设置为自动，但未运行的服务。需要判断查询 WMI 结果，如果$wmi 变量值为空，则表示自动运行的服务正常；如果有未运行自动启动的服务，则输出其数量，然后使用 foreach 语句输出其名称。此脚本的执行结果如图 18-64 所示。

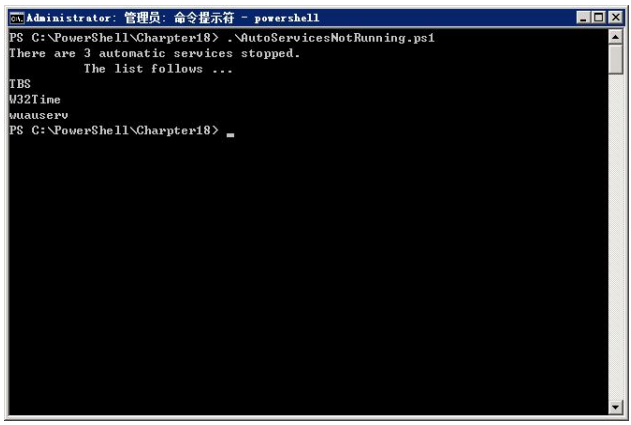


图 18-64　执行结果

18.5.3　查看硬件问题

硬件问题并不一定都和硬件有关，只要工作负荷在设计的范围内，大部分电子设备都可以使用相当长的一段时间。为了硬件正常工作，需要安装相应的驱动程序。硬件厂商会为其驱动程序添加数字签名，添加数字签名的驱动都是厂商经过大量测试后通过的，可使设备高效运转的驱动；未经签名的驱动程序可能是导致硬件问题的主要原因。

为了检查硬件是否运行厂商认证的驱动程序，创建名为“CheckSignedDeviceDrivers.ps1”的脚本，其代码如下：

```
param(
       $computer="localhost",
         [switch]$unsigned,
         [switch]$full,
         [switch]$help
 )
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor green $strIN
    Write-Host -ForegroundColor darkgreen $funline
}
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: CheckSignedDeviceDrivers.ps1
Displays a listing of device drivers that are
and whether they are signed or not
PARAMETERS:
-computer     the name of the computer
```

```
-unsigned      lists unsigned drivers
-full          lists Description, driverProviderName,
                Driverversion,DriverDate, and infName
-help          prints help file
SYNTAX:
CheckSignedDeviceDrivers.ps1 -computer munich -unsigned
Displays a listing of all unsigned drivers
on a computer named munich
CheckSignedDeviceDrivers.ps1 -unsigned -full
Displays a listing of all unsigned drivers on local
computer. Lists Description, driverProviderName,
Driverversion,DriverDate, and infName of the driver
CheckSignedDeviceDrivers.ps1 -computer munich -full
Displays a listing of all signed drivers
a computer named munich. Lists Description, driverProviderName,
Driverversion,DriverDate, and infName of the driver
CheckSignedDeviceDrivers.ps1 -help ?
Displays the help topic for the script
"@
$helpText
exit
}
if($help){ "Obtaining help ..." ; funhelp }
if($unsigned)
  { $filter = "isSigned = 'false'" ; $mode = "unsigned" }
ELSE
  { $filter = "isSigned = 'true'" ; $mode = "signed" }
$property = "Description", "driverProviderName", `
             "Driverversion","DriverDate","infName"
$wmi = Get-WmiObject -Class Win32_PnPSignedDriver `
      -computername $computer -property $property -filter $filter
funline("There are $($wmi.count) $mode drivers isted below:")
if($full)
 {
  format-list -InputObject $wmi -property `
       $property
 }
ELSE
 {
  format-table -inputobject $wmi -Property description
 }
```

其中通过 param 语句声明了 4 个参数，-computer 用于指定操作的目标计算机；-unsigned 是个 switch 参数，用于返回未曾签名的驱动的查询结果；-full 参数列出有问题的驱动程序的详细信息；-help 参数用于显示帮助文件。如果存在$unsigned 变量，则将字符串“isSigned='false'”指定给$filter 变量，该变量用于为 Get-WmiObject cmdlet 提供-filter 参数。随后在$mode 变量中保存状态信息，用于指定 WMI 查询的类型；如果不存在$unsigned 变量，则检查带签名的驱动程序。

通过数组方式指定相应的属性名，创建变量$property 并传递给 Get-WmiObject cmdlet 查询 Win32_PnPSignedDriver WMI 类；另外还使用 Count 属性统计符合条件

的驱动程序数量。如果没有签名的驱动程序，则输出之前计数器的值为空，而不是0。该脚本的执行结果如图 18-65 所示。

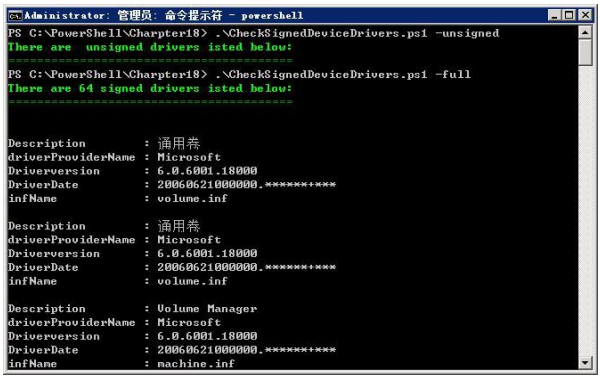


图 18-65　执行结果

需要强调的是由于 Windows Vista 和 Windows Server 2008 与 Windows XP 硬件驱动的 WMI 管理类存在差别，所以该脚本仅显示在 Windows Vista 和 Windows Server 2008 系统。

18.5.4　检查网络故障

网络故障对于用户来说是很复杂的问题，因为它可能涉及很多方面的知识，不容易查找和解决。

为检查网络故障，例 18-29 创建名为“GetActiveNicAndConfig.ps1”的脚本。

例 18-29　检查网络故障

```
param($computer = $env:computername, [switch]$full, [switch]$help)
function funline ($strIN)
{
 $strLine= "=" * $strIn.length
 Write-Host -ForegroundColor yellow $strIN
 Write-Host -ForegroundColor darkYellow $strLine
}
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: GetActiveNicAndConfig.ps1
Displays
PARAMETERS:
-computer     the name of the computer
-full         prints complete information
-help          prints help file
SYNTAX:
GetActiveNicAndConfig.ps1 -computer WebServer
Displays network adapter info and network adapter configuration info on a computer
```

```
    named WebServer
    GetActiveNicAndConfig.ps1
    Displays network adapter info and network adapter configuration info on the local machine
    GetActiveNicAndConfig.ps1 -computer WebServer -full
    Displays full network adapter info and full network adapter configuration info on a
computer named WebServer
    GetActiveNicAndConfig.ps1 -help ?
    Displays the help topic for the script
    "@
    $helpText
    exit
    }
    if($help){ "Obtaining help ..." ; funhelp }
    New-Variable -Name c_netConnected -value 2 -option constant
    $nic = Get-WmiObject -Class win32_networkadapter -computername $computer `
              -filter "NetConnectionStatus   = $c_netConnected"
    $nicConfig = Get-WmiObject -Class win32_networkadapterconfiguration `
                      -filter "interfaceindex = $($nic.interfaceindex)"
    if($full)
      {
       funline("Full Network adapter information for $($computer)")
       format-list -InputObject $nic -property [a-z]*
       funline("Full Network adapter configuration for $($computer)")
       format-list -InputObject $nicConfig -property [a-z]*
      }
    ELSE
    {
       funline("Basic Network adapter information for $($computer)")
       format-list -InputObject $nic
       funline("Basic Network adapter configuration for $($computer)")
       format-list -InputObject $nicConfig
      }
```

该脚本中使用 param 语句定义了 3 个参数，即-computer、-full 和-help，其中将-computer 变量设置为系统环境变量中的计算机名称。

funline 函数获得输入字符串的长度，并用这个长度来生成分隔线。结果保存在$strLine 变量中，然后输出带下画线的字符串。

随后的代码“New-Variable –Name c_netConnected –value 2 –option constant”创建保存已经连接网络的网络适配器数量的变量 c_netConnected，其中的数值来自 Windows 软件开发包（SDK）。使用 Get-WmiObject cmdlet 并选择 Win32_NetworkAdapter WMI 类连接到$computer 变量指定的计算机，然后查找当前已经连接的网络适配器并将结果 Management 对象保存在$nic 变量中。可以使用该对象查找相关的网络适配器配置对象，这里使用$nic.InterfaceIndex 属性。因为该属性也可以用于 Win32_NetworkAdapterConfiguration WMI 类，将结果 Management 对象保存在$nicConfig 变量中；另外还需要定义反馈信息数量的参数，在调用该脚本时提供了-full 参数获得完整的网络适配器及其配置信息。如果没有-full 参数，则仅输出每个 WMI 类的默认值。这时可以使用 Format-List cmdlet 和-InputObject 参数输

出信息。该脚本的执行结果如图 18-66 所示。

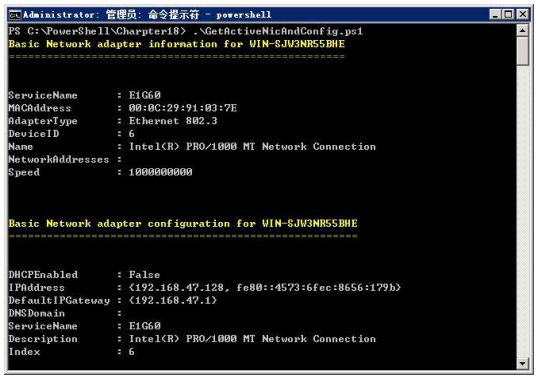


图 18-66　执行效果

需要强调的是由于管理网络适配器的 WMI 管理类是存在差异，所以该脚本仅适用于 Windows Vista 和 Windows Server 2008 系统。

18.6　证书存储

18.6.1　定位证书

在运行 Windows Vista 和 Windows Server 2008 系统的计算机中会保存大量的证书，一旦丢失，将会造成很大损失和系统安全风险。单击“开始”|“运行”选项，打开“运行”对话框。输入“certmgr.msc”按回车键即可启动系统的证书管理工具，如图 18-67 所示。

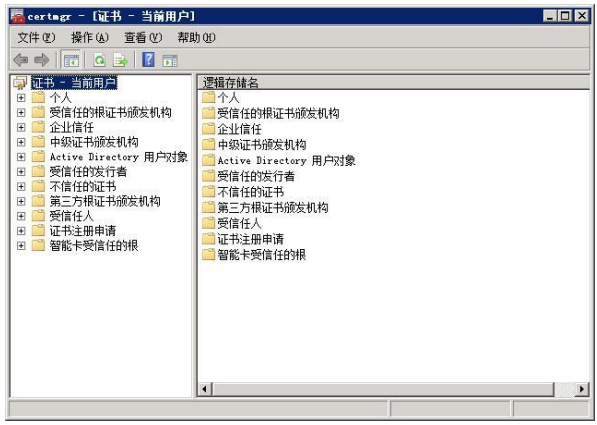


图 18-67　证书管理工具

在证书管理工具中可以看到凌乱的目录，并不能直接看出每个目录中存放的证书类型。PowerShell 提供了证书的 Provider，可以通过 Get-ChildItem cmdlet 命令获

得有关证书存储位置的相关信息，如图 18-68 所示。

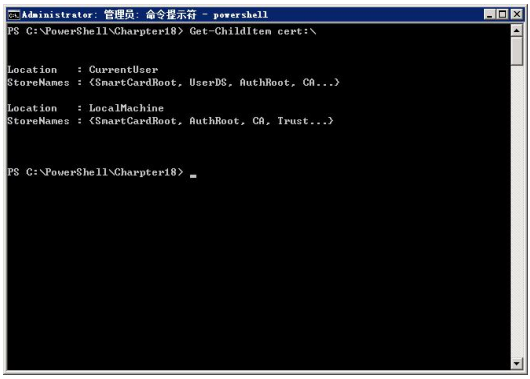


图 18-68　有关证书存储的相关信息

该命令定位到当前用户的证书存储位置，可以识别不同证书，如图 18-69 所示。

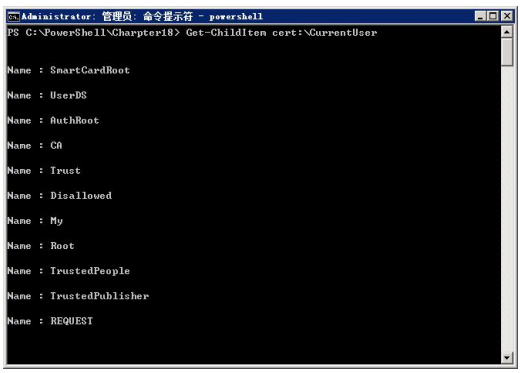


图 18-69　定位到当前用户存储的不同证书

查看颁发给特定用户的特定证书，可以使用 My 证书存储，这样就能够进入个人证书的相关目录下，如图 18-70 所示。

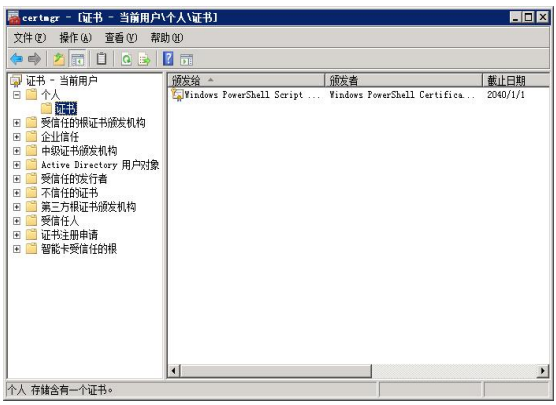


图 18-70　个人证书目录

要获得当前用户的所有个人证书列表，则使用如图 18-71 所示的命令查询。

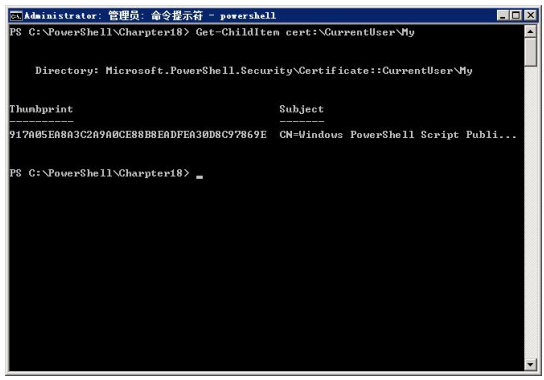


图 18-71　获得当前用户所有个人证书列表的命令

当证书颁发给用户后，通常放置在 CurrentUser\My 证书存储目录中。问题在于使用 PowerShell 定位证书时，显示的结果信息中只会包括指纹和颁发者信息，这两项信息并不足以辨别某个特定用途的证书。为了区分用途，创建名为“FindCertificates.ps1”脚本。其中使用 EnhancedKeyUses 的 FriendlyName 属性，这个属性来自 Sytem.Security.Cryptography.X509ExtensionCollection 的.NET 框架类，该脚本的代码如下：

```
param($use, [switch]$help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: FindCertificates.ps1
Finds certificates of a particular use on the local machine
PARAMETERS:
-use        the purpose for the certificate ex: code signing,
            client authentication, smart card logon etc.
-help       prints help file
SYNTAX:
FindCertificates.ps1
Gets a listing of all certificates in the my store FindCertificates.ps1 -use "digital signature"
Gets a listing of certificates in my store that provide a digital signature on local computer
FindCertificates.ps1 -use "code signing"
Gets a listing of certificates in my store that provide code signing support
FindCertificates.ps1 -help
Prints the help topic for the script
"@
 $helpText
 exit
}
if($help) { "Printing help now..." ; funHelp }
if(!$use) { "A use is required..." ; funHelp }
$myCert = Get-ChildItem cert:\CurrentUser\My
ForEach( $cert in $myCert)
 {
   $certExt = $cert.get_extensions()
    Foreach( $ext in $certExt )
      {
        foreach( $name in $ext.enhancedKeyUsages )
```

```
            {
             if($name.friendlyname -match $use)
                {
                 "Certificates that match $use"
                 "$($name.friendlyname) certificate: `
                 `n$($cert.thumbprint) `n$($cert.subject)`n"
                }
            }
        }
    }
```

此脚本开始处通过 param 语句从命令行收集可以控制脚本功能的参数，并定义了-use 参数，用于收集想要查看的特定证书的目标信息。取决于证书的用途，获得的值可能不同，如代码签名、智能卡验证或数字签名等。由于执行正则表达式匹配，所以不需要输入完全的友好名称。

该脚本检查用户的输入是否包括$use 变量，如果不存在，表示在命令行下没有提供相应的参数，则输出状态信息并调用 funhelp 函数输出帮助信息。接下来获取 My 证书存储目录内所有证书对象的集合，并保存在$mycert 变量中。为此可以使用 Get-ChildItem 指定目录到名为“cert:\”的 PS 驱动器，然后在 CurrentUser\My 证书存储目录内浏览。由于在该脚本中将证书存储目录中的证书对象保存到$mycert 变量中，因此需要遍历整个集合。随后使用 ForEach 语句枚举和遍历，分别为每个独立的证书对象调用 get-extensions()方法。这样可以返回一系列扩展对象，并将其保存在$certext 变量中。然后遍历扩展对象，将$ext 变量作为枚举对象使用。每个扩展对象由两个属性组成，但在该脚本中仅查找 FriendlyName 属性。为此使用$name 变量再次遍历整个集合，如果找到与保存在$use 变量中字符串匹配的正则表达式，则输出一个包括属性名的字符串。

（1）列出证书

为列出特定证书存储目录中的证书，创建名为“ListCertificates.ps1”脚本。在其中使用位于 System.Security.Cryptography.X509Certificates.ps1 命名空间中的.NET Framework 的 X509Store 类，借助这个类可以通过使用 New-Object cmdlet 创建其实例。

例 18-30　列出证书

```
param($store="my", [switch]$listStores, [switch]$help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: ListCertificates.ps1
Lists certificates on the current machine
PARAMETERS:
-store      the certificate store to search
-help       prints help file
SYNTAX:
ListCertificates.ps1
Gets a listing of all certificates in the my store
```

```
ListCertificates.ps1 -store "authroot"
Gets a listing of certificates in authroot store on local computer
ListCertificates.ps1 -store "my"
Gets a listing of certificates in my store on local computer
ListCertificates.ps1 -help
Prints the help topic for the script
"@
 $helpText
 exit
}
Function funstore()
{
 write-host -foregroundcolor green "Listing currentuser stores:"
  Get-ChildItem cert:\CurrentUser
  write-host -foregroundcolor green "Listing localmachine stores:`n"
  Get-ChildItem cert:\LocalMachine
  exit
}
if($help) { "Printing help now..." ; funHelp }
if($liststores) { funstore }
new-variable -name userStore -value "currentUser" -option readonly
$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
$objStore = new-object $crypto $store
$objstore.Open("Readonly")
$colcerts = $objstore.Certificates
Write-Host -ForegroundColor blue    `
 "   There are $($colcerts.count) certificates in the $store store.
  They are listed below:
 "
foreach($cert in $colCerts)
 {
  "FriendlyName:   $($cert.FriendlyName)"
  "Serialnumber: $($cert.SerialNumber)"
  "Thumbprint: $($cert.thumbprint)"
  "Subject: $($cert.subject)`n"
 }
$objstore.Close()
```

该脚本开始使用 param 语句定义了 3 个参数，其中-store 指定提供证书列表的证书存储目录，默认值为 My 证书存储；-liststore 为 switch 类型，指定提供本地计算机中所有证书存储目录的详细列表；-help 为 switch 类型，用于显示帮助信息。

随后创建一个 funstore()的函数，用于用户输入-ListStores 参数时提供本地计算机中所有证书存储的列表。首先使用 Write-Host cmdlet 用绿色输出当前用户存储位置的属性名称信息，然后使用 Get-ChildItem cmdlet 并指向 cert:\这个 PSDrive 中的当前用户的存储位置，同时为遍历 LocalMachine 证书位置执行同样的操作。

该脚本首先检查是否存在$help 变量，如果存在，则调用 funhelp()函数显示帮助信息；另外检查是否存在$ListStores 变量，如果存在，则调用前面定义的 funstore()函数显示本机中使用过的证书存储目录。

该脚本通过 New-Variable cmdlet 声明了一个名为“userstore”只读变量并设其值为 CurrentUser，然后使用-option 参数以确保该变量为只读。创建名为“$crypto”的变量，将其值设置为代表 X509Store 的.NET 类的绝对位置。这样使程序的可读性更好，因为类名称和命名空间名称的组合可能会很长。随后创建一个 X509Store 类的实例，将其路径及$store 变量中包括的存储位置传递给该类，并将返回的对象保存在$objstore 变量中。通过使用 readonly 关键字的方式以只读模式打开证书存储，将其提供给 Open()方法。为了创建证书存储目录的所有证书的集合，可以查询 Certificates 属性并将其结果保存在$colcers 变量中。

为了创建证书列表的标题，该脚本使用 Write-Host cmdlet，并将前景色-foreground 参数设置为蓝色。随后输出一条信息，并用子表达式获得集合中证书的数量。为此在脚本中在$colcerts.count 之前添加一个美元符（$），并用括号包括除了开头$符以外的所有内容，如$($colcerts.count)。这样即可知道证书总数，而不必逐个展开对象的名称。

因为希望得到证书集合，所以必须使用 foreach 语句并使用$cert 变量作为枚举对象。对于集合中的每个证书，为需要查询的每个属性使用子表达式，输出属性后关闭证书存储。该脚本的执行结果如图 18-72 和图 18-73 所示。

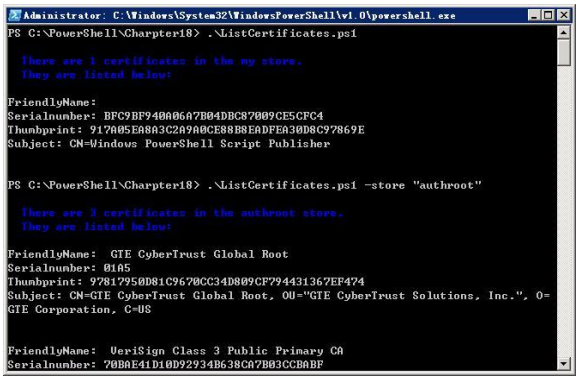


图 18-72　获取个人证书存储区和授权根存储区的证书信息

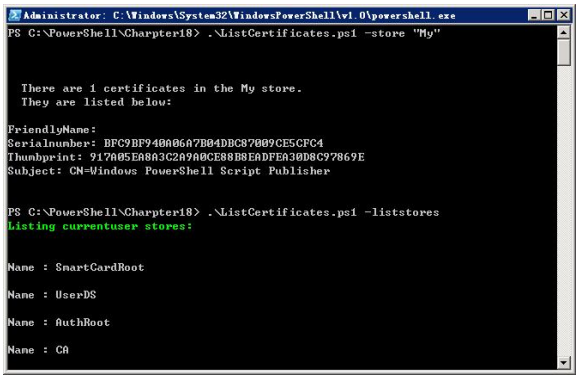


图 18-73　获取当前用户存储目录的证书/和本机所有证书存储目录

（2）定位过期的证书

随着数字证书使用范围的越来越广，经常会发生证书过期的情况。很多用户可能遇到过在连接到网上银行、在线支付一类的网站，浏览器提示该网站所使用的证书已过期，是否要继续浏览，继续存在安全风险的。作为排错的方法，用户可能需要快速定位到过期的证书。为此创建名为“FindExpiredCertificates.ps1”脚本，如例 18-31 所示。

例 18-31　定位过期的证书

```
param(
       $store,
       [switch]$listcu,
       [switch]$listlm,
       [switch]$help
      )
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: FindExpiredCertificates.ps1
Finds expired certificates on the local machine
PARAMETERS:
-store      the certificate store on the computer
-help       prints help file
SYNTAX:
FindExpiredCertificates.ps1
Gets a listing of expired certificates in the my store of the currentuser
FindExpiredCertificates.ps1 -store "currentuser\my"
Gets a listing of expired certificates in the my store of the currentuser
FindExpiredCertificates.ps1 -store "currentuser\smartcardroot"
Gets a listing of expired certificates in the smartcardtoot store of the currentuser
FindExpiredCertificates.ps1 -listcu
Gets a listing of certificate stores for the currentuser
FindExpiredCertificates.ps1 -listlm
Gets a listing of certificate stores for the localmachine
FindExpiredCertificates.ps1 -help
Prints the help topic for the script
"@
 $helpText
 exit
}
if($help)    { "Printing help now..." ; funHelp }
if($listcu) {
               "Certificate stores in currentuser"
               get-childitem cert:\currentuser ; exit
              }
if($listlm) {
               "Certificate stores in localmachine"
               get-childitem cert:\localmachine ; exit
              }
if(!$store) {
               $store = "currentuser\my"
               "Using default store: $store"
```

```
                "See $($myinvocation.mycommand) -help" `
                + " for additional examples"
               }
$currentDate = Get-Date
$colcert = Get-ChildItem cert:\$store
Write-host -foregroundcolor cyan "Expired Certificates in $store"
foreach($cert in $colcert)
{
 if($cert.notafter -lt $currentDate)
   {
    Write-host `
      "
       $($cert.thumbprint) `t $($cert.Notafter)
      "
   }
}
```

该脚本使用 param 语句定义了 4 个命令行参数，其中-store 用于指定要访问的证书存储目录，这是一个必须的参数；-listcu 和-listlm 为 switch 类型，分别指定列出当前用户位置和本地计算机中的证书存储目录；-help 指定显示帮助信息。

如果存在$listcu 变量，表示在运行脚本时使用了-listcu 参数，则输出一条简短的状态信息。然后使用 Get-ChildItem cmdlet 产生当前用户位置的证书存储列表，最终退出；如果存在$listlm 参数，表示在运行脚本时使用了-listlm 参数，则输出一条状态信息并使用 Get-ChildItem cmdlet 产生本机中所有证书存储目录的列表。-store 参数主要用于控制需要对哪个证书存储目录查询过期证书，如果没有该参数，那么脚本会默认查询当前用户的 My 证书存储目录。随后输出信息告知用户，输出的内容是在使用默认值情况下的输出。然后使用$myinvocation.mycommand 命令输出正在运行的脚本名称，并建议使用-help 参数的获取更详细的信息。

接下来使用 Get-Date cmdlet 活的 datetime 对象，然后将其保存在$currentdate 变量中。并使用 Get-ChildItem cmdlet 获得由$store 变量值指定的证书存储中所有证书的集合，使用$colcert 变量用于保存证书的集合。最后使用 Write-Host cmdlet 指定前景色-foreground 参数输出青色的信息，并使用 foreach 语句遍历整个证书集合，以$cert 变量作为枚举对象。一旦在$cert 变量中保存了每个单独证书存储目录的信息，还需要查看 NotAfter 的属性以判断其值是否小于保存在$currentdata 变量中的当前时间。如果小于，则输出该证书的指纹和过期时间。该脚本的执行结果如图 18-74 所示，能够看到当前计算机中没有过期的证书。

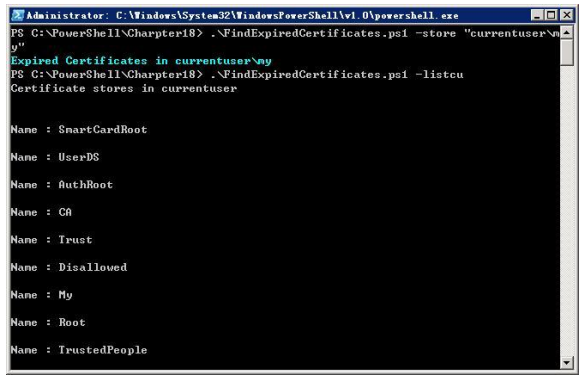


图 18-74　执行效果

（3）识别即将过期的证书

为用户颁发的证书的有效期限一般是 1~2 年，有必要使用脚本提前检查证书的过期时间，在证书失效之前提示用户。为此创建名为“FindCertificatesAboutToExpire.ps1”的脚本，其代码如例 18-32 所示。

例 18-32　识别即将过期的证书

```
param(
       $store,
       $days=30,
       [switch]$listcu,
       [switch]$listlm,
       [switch]$help
      )
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: FindCertificatesAboutToExpire.ps1
Finds certificates about to expire with in a certain
number of days on the local machine
PARAMETERS:
-store      the certificate store on the computer
-days       number of days in the future to evaluate for
            certificate expiration
-help       prints help file
SYNTAX:
FindCertificatesAboutToExpire.ps1
Gets a listing of certificates about to expire within 30 days
in the my store of the currentuser
FindCertificatesAboutToExpire.ps1 -days 45
Gets a listing of certificates about to expire within 45 days
in the my store of the currentuser
FindCertificatesAboutToExpire.ps1 -store "currentuser\my" -days 60
Gets a listing of certificates about to expire within 60 days
in the my store of the currentuser
FindCertificatesAboutToExpire.ps1 -store "currentuser\smartcardroot"
Gets a listing of certificates about to expire within 30 days
in the smartcardroot store of the currentuser
FindCertificatesAboutToExpire.ps1 -listcu
Gets a listing of certificate stores for the
currentuser
FindCertificatesAboutToExpire.ps1 -listlm
Gets a listing of certificate stores for the
localmachine
FindCertificatesAboutToExpire.ps1 -help
Prints the help topic for the script
"@
 $helpText
 exit
}
if($help)    { "Printing help now..." ; funHelp }
```

```
if($listcu) {
              "Certificate stores in currentuser"
              get-childitem cert:\currentuser ; exit
             }
if($listlm) {
              "Certificate stores in localmachine"
              get-childitem cert:\localmachine ; exit
             }
if(!$store) {
              $store = "currentuser\my"
              "Using default store: $store"
              "See $($myinvocation.mycommand) -help" `
              + " for additional examples"
             }
$currentDate = (Get-Date).adddays($days)
$colcert = Get-ChildItem cert:\$store
Write-host -foregroundcolor cyan "Certificates in $store that" `
                                  "expire in $days days"
foreach($cert in $colcert)
{
 if($cert.notafter -lt $currentDate)
   {
    Write-host `
      "
       $($cert.thumbprint) `t $($cert.Notafter)
      "
   }
}
```

该脚本首先使用 param 语句定义了 5 个参数，其中-store 是必须的；-day 的默认值为 30 天；-listcu 用于列出当前用户位置下的所有可用证书存储区域；-listlm 可以为本机位置下列出所有可用证书存储区域；-help 可以输出帮助信息。

如果用户未提供必需的参数-store，则不存在$store 变量，该脚本会默认查询 My 证书存储目录并提示用户其他可用选项。使用$myinvocation.mycommand 命令输出脚本名称，获取脚本名称使用了一个子表达式$($myinvocation.mycommand)。

检查参数后开始处理脚本中逻辑控制部分，首先为 System.DateTime 的.NET Framework 对象创建实例，然后使用 adddays()方法将天数添加到当前日期中。将生成的日期保存在$currentdate 的变量中，并通过使用 Get-Children cmdlet 获得证书的集合。接下来为了便于识别并输出前景色不同的标题，使用 foreach 语句遍历证书集合，使用$cert 变量作为枚举变量确保操作针对之前确定的集合位置。然后检查每个证书的 NotAfter 属性，这样即可知道其过期日期。如果此日期小于保存在$currentdata 变量中的日期，则输出该证书的指纹和过期日期。该脚本的执行结果如图 18-75 所示，能够看到默认的 30 天内没有要过期的证书。为了能够找到一个将要过期的证书，这里将天数设置到 60000 天，搜索到一个将在 2040 年过期的证书。

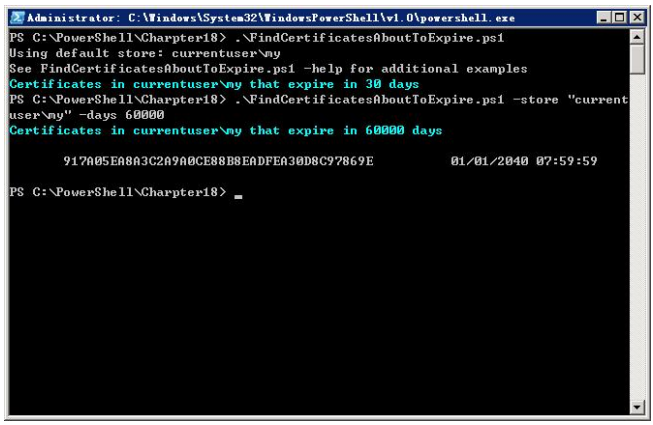


图 18-75　执行结果

18.6.2　管理证书

证书管理包括查看、导入及删除证书。

（1）查看证书

查看证书可以确保证书是需要的，而且来源合法。为此创建名为“InspectCertificate.ps1”的脚本，在其中使用位于 Security.Cryptography.X509Certificates 的命名空间中的.NET Framework 的 X509Certificate 类。要查看的属性与图 18-76 所示的证书管理工具中的属性相同。

图 18-76　证书管理工具中的证书属性

```
param($cert, [switch]$help)
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: InspectCertificate.ps1
```

```
Finds certificates of a particular use on the local machine
PARAMETERS:
-cert        the full path to the certificate to inspect
-help        prints help file
SYNTAX:
InspectCertificate.ps1
Generates an error that a certificate is required
InspectCertificate.ps1 -cert "c:\fso\filerecovery.cer"
Inspects a certificate called filerecovery in the c:\fso
directory. This certificate could be DER encoded or base -64
encoded .cer file.
InspectCertificate.ps1 -help
Prints the help topic for the script
"@
 $helpText
 exit
}
if($help) { "Printing help now..." ; funHelp }
if(!$cert) { "A certificate is required..." ; funHelp }
$objCert=[security.cryptography.x509certificates.x509certificate]"$cert"
"HashString: $($objCert.GetCertHashString())"
"EffectiveDate: $($objCert.GetEffectiveDateString())"
"ExpirationDate: $($objCert.GetExpirationDateString())"
"HashCode: $($objCert.GetHashCode())"
"KeyAlgorithm: $($objCert.GetKeyAlgorithm())"
"KeyAlgorithmParameters: $($objCert.GetKeyAlgorithmParametersString())"
"Name: $($objCert.GetName())"
"SerialNumber: $($objCert.GetSerialNumberString())"
"Cert: $($objCert.ToString())"
"Issuer: $($objCert.Issuer)"
"Subject: $($objCert.Subject)`n"
"PublicKey: $($objCert.GetPublicKeyString())`n"
"RawCertData: $($objCert.GetRawCertDataString())`n"
```

该脚本开始使用 param 语句定义了两个必要的参数，其中-cert 指定要查看证书的完整路径和名称；-help 如前所述。接下来使用 if 语句查看是否存在$help 变量，如果存在，则调用定义的 funhelp()函数，在脚本中为了代码的可读性将两个语句用分号隔开放在同一行中。随后使用非操作（!) 检查是否存在$cert 变量，如果不存在，则调用 funhelp ()函数输出帮助。

为创建与证书的连接，使用 X509Certificate 这个.NET Framework 类。在.NET Framework 中为了便于创建类的实例，可以使用与类型转换相同的方法来创建类的实例。例如，如果希望创建字符串，那么可以使用下面的语法转换为 System.String 类型：

```
[string]"This is a string"
```

如果希望创建一个 X509Certiricate 的实例，则可以使用类似的语法：

```
$objCert = [Security.Cryptography.X509Certificates.X509Certificate] "$cert"
```

连接到证书可以使用$cert 变量中包括的证书对象行类型，并指向.NET Framework 的 X509Certificate 类，然后将新对象保存在$objCert 变量中。该脚本的

其余部分使用子表达式输出证书属性的代码，执行结果如图 18-77 所示。

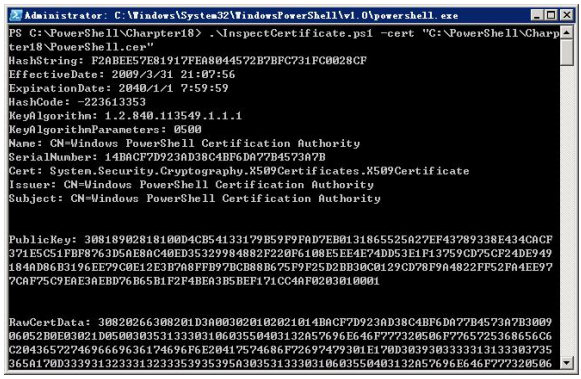


图 18-77　执行结果

（2）导入证书

导入证书到证书存储区域中可以打开如图 18-78 所示“证书导入向导”对话框并按提示操作。

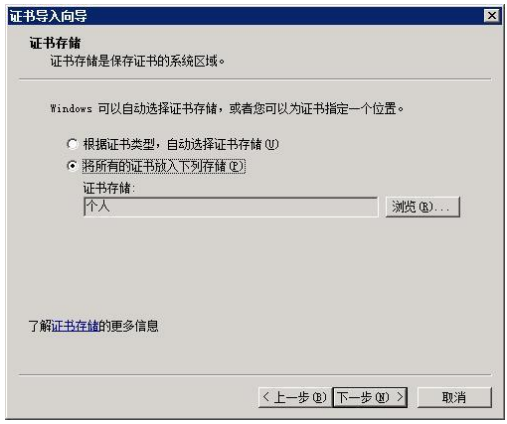


图 18-78　“证书导入向导”对话框

为使用 Windows PowerShell 脚本导入证书，创建名为“ImportCertificate.ps1”的脚本，其代码如下：

```
param(
      $cert,
      $store = "my",
      [switch]$liststores,
      [switch]$help
     )
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: ImportCertificate.ps1
Imports a certificate into a certificate store
PARAMETERS:
```

```
    -cert          path of certificate to import
    -store         the certificate store on the computer
    -liststores lists certificate stores on local machine
    -help          prints help file
    SYNTAX:
    ImportCertificate.ps1
    Prints error message a certificate is required, and displays help
    ImportCertificate.ps1 -cert "c:\fso\mycert.pfx"
    Imports a certificate stored in the c:\fso folder named
    mycert.pfx into the my store of the currentuser
    ImportCertificate.ps1 -store "my" -cert   "c:\fso\mycert.pfx"
    Imports a certificate stored in the c:\fso folder named mycert.pfx into the my store of the
currentuser
    ImportCertificate.ps1 -store "smartcardroot"
    -cert "c:\fso\mycert.pfx"
    Imports a certificate stored in the c:\fso folder named
    mycert.pfx into the smartcardroot store of the currentuser
    ImportCertificate.ps1 -liststores
    Gets a listing of certificate stores for the currentuser
    ImportCertificate.ps1 -help
    Prints the help topic for the script
    "@
     $helpText
     exit
    }
    Function funstore()
    {
     write-host -foregroundcolor green "Listing currentuser stores:"
      Get-ChildItem cert:\CurrentUser
      write-host -foregroundcolor green "Listing localmachine stores:`n"
      Get-ChildItem cert:\LocalMachine
      exit
    }
    if($help)   { "Printing help now..." ; funHelp }
    if($liststores) { funStore }
    if(!$cert) {
                  "A certificate path is required..." ;
                  funhelp
                }
    new-variable -name userStore -value "currentUser" -option readonly
    $crypto = "System.Security.Cryptography.X509Certificates.X509Store"
    $objStore = new-object $crypto $store, $userStore
    $objstore.Open("ReadWrite")
    $objstore.Add($cert)
    $objstore.Close()
```

在该脚本开始处使用 param 语句定义了 4 个参数，其中-cert 指定保存要导入的证书的路径；-store 指定导入证书的目标存放证书存储目录，默认值为 My 证书存储目录。对于证书管理器来说，该默认值代表个人证书存储；-liststores 指定列出当前用户命名空间下的可用证书存储目录；-help 显示帮助信息。随后定义 funstore 函数用于输出对当前用户证书存储目录和本机证书存储目录。

检查-help 参数，判断是否存在$help 变量。如果存在，则输出提示信息的同时调用 funhelp()函数；检查-liststores 参数，如果存在$liststores 变量，则调用 funstore()函数；检查是否存在$cert 变量，如果用户未使用-cert 参数且未指定-help 或-liststores 参数，则输出一条错误信息提示用户需要指定证书，随后调用 funhelp()函数。

接下来声明两个变量，一是$userstore 变量，将其值设置为 CurrentUser，然后设置为只读；二是$crypto，将其值设置为下面要使用的 X509Store 类的完整命名空间字符串。随后使用 New-Object cmdlet 创建 X509Store 类的实例，传递保存在$cypto 变量中的字符串、位于$store 变量中的证书存储位置，以及包括在$userstore 变量中的证书存储目录，然后将 X509Store 对象保存在$objstore 变量中。

在创建 X509Store 对象后，使用 Open()方法并选择读写模式。调用 Add()方法将其传递给$cert 变量，其中包括证书对象的实例。在完成后务必调用 Close 方法关闭对象，该脚本的执行结果如图 18-79 所示。

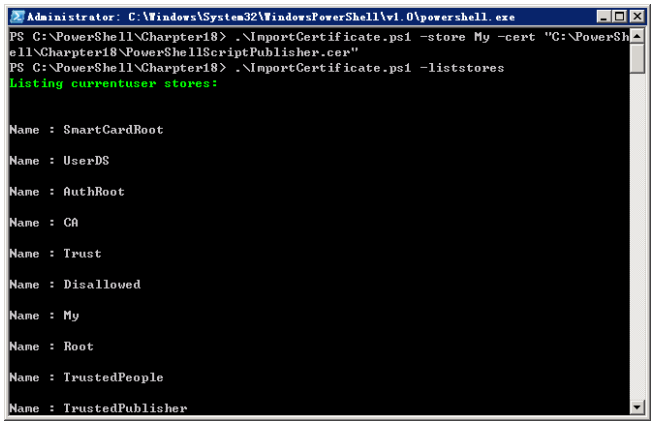


图 18-79　将证书导入到当前用户证书存储区域

（3）删除证书

可以使用证书管理工具删除过期或者不再信任的证书。为便于删除大量的证书，创建名为“DeleteCertificate.ps1”的脚本，其代码如例 18-33 所示。

例 18-33　删除大量证书

```
param(
      $cert,
      $store = "my",
      [switch]$listcerts,
      [switch]$help
     )
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: DeleteCertificate.ps1
Removes a certificate from a certificate store
PARAMETERS:
```

```
    -store        the certificate store on the computer
    -cert         certificate to delete
    -listcerts    lists certificates in specified store
    -help         prints help file
    SYNTAX:
    DeleteCertificate.ps1
    Prints error message a certificate is required, and displays help
    DeleteCertificate.ps1 -cert "B67BAFECA1E77B8F3AEAB8EB9054D5D31C3C0A03"
    Removes a certificate with thumbprint of
B67BAFECA1E77B8F3AEAB8EB9054D5D31C3C0A03 from the my store of the currentuser
    DeleteCertificate.ps1 -store "my" -cert "OU=EFS File Encryption Certificate"
    Removes a certificate with subject of OU=EFS File Encryption Certificate from the my store
of the currentuser
    DeleteCertificate.ps1 -store "smartcardroot" -cert
"E47F375796238DB54CB70DA7A5E88F79"
    Removes a certificate with the serial number of
    E47F375796238DB54CB70DA7A5E88F79 from the smartcardroot store of the currentuser
    DeleteCertificate.ps1 -listcerts
    Gets a listing of certificates for the my store of the currentuser
    DeleteCertificate.ps1 -help
    Prints the help topic for the script
    "@
     $helpText
     exit
    }
    Function funcert()
    {
     $crypto = "System.Security.Cryptography.X509Certificates.X509Store"
     $objStore = new-object $crypto $store, $userStore
     $objstore.Open("ReadWrite")
     $colcerts = $objstore.Certificates
     Write-Host -ForegroundColor blue
      "
        There are $($colcerts.count) certificates in the $store store.
        They are listed below:
      "
     foreach($cert in $colCerts)
      {
       "FriendlyName:   $($cert.FriendlyName)"
       "Serialnumber: $($cert.SerialNumber)"
       "Thumbprint: $($cert.thumbprint)"
       "Subject: $($cert.subject)`n"
      }
     $objstore.Close()
     exit
    }
    Function findcert($key)
    {
     $crypto = "System.Security.Cryptography.X509Certificates.X509Store"
     $objStore = new-object $crypto $store, $userStore
     $objstore.Open("ReadWrite")
     $colcerts = $objstore.Certificates

     foreach($cert in $colCerts)
```

```
    {
     if($cert.thumbprint -match $key)    { $global:mycert = $cert }
     if($cert.serialnumber -match $key) { $global:mycert = $cert }
     if($cert.friendlyname -match $key) { $global:mycert = $cert }
     if($cert.subject -match $key)       { $global:mycert = $cert }
    }
   }
new-variable -name userStore -value "currentUser" -option readonly
$global:mycert = $null
if($help)   { "Printing help now..." ; funHelp }
if($listcerts) { "Listing certificates in $store" ; funcert }
if(!$cert) {
              "A certificate is required..." ;
              funhelp
            }
Findcert($cert)
$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
$objStore = new-object $crypto $store, $userStore
$objstore.Open("ReadWrite")
$objstore.remove($mycert)
$objstore.Close()
```

在脚本开始使用 param 语句定义了 4 个参数，其中-cert 是必须的，用于指定计算机上要删除证书；-store 的默认值为 My，用于指定要删除的证书所在的证书存储目录；-listcerts 可以上脚本列出所选证书存储区域中所有的证书；-help 用于显示与脚本相关的帮助信息。

该脚本创建的 funcert()函数内部定义了用于保存代表 X509Store 类的命名空间和类名称字符串的名为“$crypto”的变量。随后使用 New-Object cmdlet 创建 X509Store 类的新实例，这个类的构造函数需要预先指定存储位置及其下的证书存储目录名称。因此可以使用在$store 和$userstore 变量值，然后将返回的 X509Store 对象保存在$objstore 变量中。接着使用 Open()方法打开证书存储，并以读写模式打开该证书存储目录使脚本可以访问。查询 Certificates 属性并利用查询的结果创建证书集合，然后保存在$colcerts 变量中，在函数最后使用 Write-Host cmdlet 输出证书列表的标题。遍历包括在$colcerts 变量中的集合并使用$cert 变量作为枚举对象，以处理用户希望看到的每个独立的证书。将独立的证书保存在$cert 变量中，并输出这些证书的友好名称、序列号、指纹，以及颁发者名称。处理整个集合后关闭存储目录对象，并退出函数。

findcert()函数用于搜索特定证书，如果找到，则返回的证书对象保存到脚本的全局变量$mycert 中，该函数和前面 funcert()函数中创建 X509Store 实例的方法相同。findcert()和 funcert()函数在对$colcerts 变量中包括的集合枚举循环处有所差别，如果可以匹配$cert 变量中包括的证书对象指纹、序列号、友好名称及颁发者信息属性，则 findcert()函数传递的$key 变量将$cert 变量中的该证书对象保存在全局变量$mycert 中。

创建脚本中的主要函数后，该脚本中使用 New-Variable cmdlet 创建名为

“$userstore”的只读变量，并为其赋值为 CurrentUser。然后将$mycert 变量初始化为全局变量，并赋值为$null。

接下来检查命令行接受的参数，首先检查-help 参数。如果存在$help 变量，则输出一条提示信息，并调用 funhelp()函数；如果存在$listcerts 变量，则调用 funcert()函数。最后检查是否存在$cert 变量，如果不存在，则输出一条错误信息并调用 funhelp()函数。

在确定所有命令行参数满足要求后调用 findcert()函数，并将保存在$cert 变量中的证书名称传递过来。在获得证书对象并将其保存在$mycert 变量中后创建一个 X509Store 类的实例，打开证书存储目录。将保存在$mycert 变量中的证书对象传递过来，并调用 Remove()方法删除其中对应的证书，最后关闭证书存储目录。该脚本的执行结果如图 18-80 所示，执行后可以在证书管理工具中看到相应的证书已经删除。

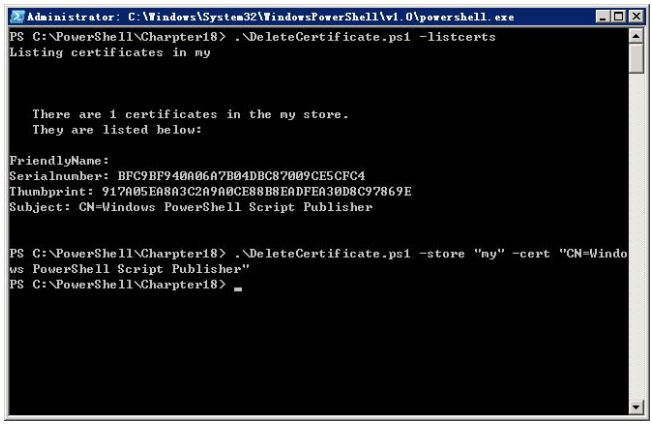


图 18-80　执行结果

18.7　小结

本章中介绍的如下主要内容基本覆盖了 PowerShell 的桌面计算机的常见操作。

（1）Windows Vista 和 Windows Server 2008 的事件日志包括使用 Get-EventLog cmdlet 生成可用的事件日志清单、使用同一个 cmdlet 读取不同的事件日志、搜索事件日志、将信息写入事件日志，以及创建自定义事件日志。

（2）管理企业环境中桌面计算机的磁盘和文件，包括使用脚本收集磁盘卷的配置信息及通过 WMI 性能计数器获取系统的运行状态。

（3）使用 PowerShell 操作用户和组、设置计算机桌面、检查屏幕保护程序并判断其是否安全，以及获取当前电源配置信息并设置有关选项等电源配置操作。

（4）管理网络，包括设置和 TCP/IP 堆栈有关的选项；通过不同脚本提供网络

适配器的状态信息，网络适配器的连接状态及属性；设置静态 IP、启动 DHCP 及配置 DNS 服务器；获取防火墙设置信息并设置有关选项以启用远程管理，以及远程共享文件等。

（5）解决 Windows Vista 和 Windows Server 2008 的常见问题，包括设置引导信息、查找启动服务和自启动服务未正常启动的方法、服务的依存性、识别未经签名的设备驱动程序，以及查找当前活动网卡的配置信息。

（6）多种管理证书的方法，包括通过搜索证书存储目录、定位特定的证书、使用.NET Framework 类列出特定命名空间下的所有证书、定位已过期和即将过期的证书、查看证书、导入证书，以及删除证书等。